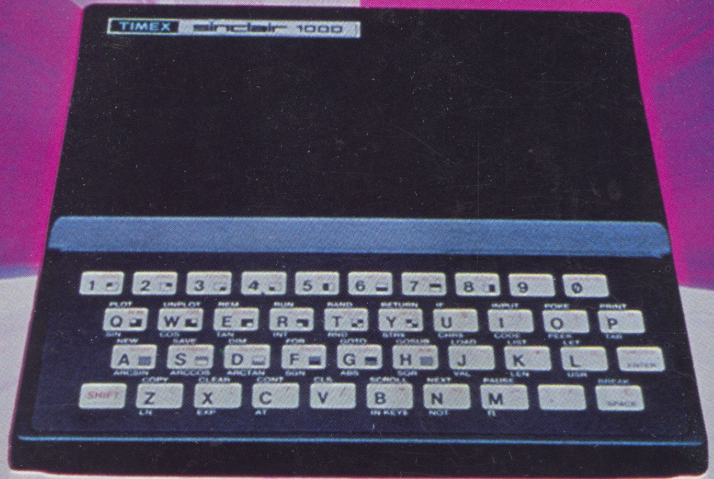
YOUR TIMEX SINCLAIR 1000™ AND ZX81™



DOUGLAS HERGERT

SYBEX

6

Your Timex Sinclair 1000 and ZX81

Your Timex Sinclair 1000 and ZX81

Douglas Hergert

SYBEX

Berkeley • Paris • Düsseldorf

Book design by Valerie Brewster
Photography by Eye to Eye Images, San Francisco
Cover Design by Jack Morton Productions, San Francisco

Library of Congress Card Number: 82-62360
ISBN 0-89588-099-7
First Edition 1983
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

To Mme Berthe Badji

Contents

Acknowledgements

The following people helped develop an idea into a manuscript, and then the manuscript into a book: James Compton, Joel Kreisman, and Rudolph Langer, editorial and technical support; Bret Rohmer, production; Elaine Foster and Donna Scanlon, word processing; Ingrid Owen, Sharon Leong and Margaret Cusick, paste-up; Valerie Brewster, typesetting; and Hilda van Genderen and Cheryl Wilcox, proofreading.

I would also like to extend a personal note of thanks to the entire staff at Sybex.

D.H.

Introduction: Entering the Computer Age

You've just arrived home with your new computer—the Timex/Sinclair 1000™. This is a moment of great excitement and enthusiasm; you've finally entered the computer age. No longer will you sit back in quiet ignorance as your friends talk about their personal computers. Now you'll be part of the club. You reflect happily on the small investment you had to make, compared to the thousands of dollars others have spent for their computers.

You plug in your computer, attach it to the TV set, and note with great satisfaction that the computer is responding as expected.

At this point there are two very different scenarios that might describe your subsequent computer experience. The first scenario, which ends in disillusionment and disappointment, is completely unsatisfactory. The second scenario, which you have every reason to expect, is the one that this book will lead you through. Let's consider these two scenarios.

In scenario one you begin pecking at the keyboard and watching what happens on the screen; at first you marvel at the power you seem to have over the action of the computer. But gradually you begin to realize that you don't really *understand* what's happening. True, the computer does something each time you press the keyboard, but what is it doing? Perhaps this is going to be a little more work than you had anticipated. Never mind, you think, the manual will explain it all. So you open up the little spiral-bound book that came with the computer, and you begin reading. After a paragraph or two you begin feeling a little uneasy. After several pages you feel genuinely lost. Somehow this book seems to be addressed to people who already know more than you do. You sense that it must certainly be filled with useful information for a person with a little experience—as in fact it is—but how do you get started? You return to the computer, hopefully press a few more keys, thinking that it will all come to you in a moment. But the moment never arrives. Somehow all your initial enthusiasm is lost, replaced by an unexpected feeling of confusion and disappointment. You unplug the computer, put it in the closet, go out to mow the lawn . . . Two or three months later you notice the computer, still sitting there in the closet, with a thin layer of dust over the keyboard. You have become fond of telling people that you *own* a computer, but, alas, you always have to find a way of changing the subject when your friends begin asking detailed questions. . .

In scenario two, you approach the computer more systematically. You realize right away that there's a lot to learn about your new computer, and the more you learn the greater your enthusiasm becomes. You start out by mastering the Timex Sinclair keyboard. You learn to enter commands, and you find out what each command means to the computer. You type a complete program in BASIC, the computer language built into the T/S 1000, and impress yourself with the amusing results. The experience becomes more fascinating each time you sit down at the computer; you learn about graphics, calculations, string handling. . . and before you know it you are an accomplished BASIC programmer, impressing not only yourself, but also your family and friends, with all the things you can do on your new computer. After several weeks you realize that you are more than a computer owner—much more importantly, you are a computer *user.*

This book, *Your Timex Sinclair 1000 and ZX81,* was written to make sure you experience the second of these two scenarios. If you are completely new to computers, completely confused by all the new computer vocabulary that everyone else seems to be picking up automatically, then this book is for you. *Your Timex Sinclair 1000 and ZX81* starts out at the very beginning of the story, and guides you carefully through all the steps you need to follow to become a satisfied computer user.

Chapter 1—*The Cast of Characters*—describes the *hardware*—all of the equipment you'll need to make the most of your computer. It also gives you careful, step-by-step instructions for connecting equipment to your computer. Finally, this chapter discusses your own role in making the computer work for you, and the skills you will begin developing as you use your computer.

Chapter 2—*The First Act: Enter Your Program*—is an introduction to *software,* the instructions and *programs* that you prepare in order to guide your computer through specific tasks. You will start out learning exactly how to use the T/S 1000 keyboard to communicate with your computer. You will enter an entire program, run it, and, in the process, get your first glimpse of the power of the BASIC programming language. Finally, you'll learn how to *save* a program permanently on a cassette tape, so that you can use the program again and again. This chapter ends with a consumer's guide to *buying* software, and a discussion of the pros and cons of buying vs. writing your own programs.

Chapter 3—*The Plot Thickens*—starts you out in a quick and easy course in BASIC. You'll learn how to instruct the computer to repeat certain tasks over and over; to make decisions based on current information; and to display the results on the TV screen where you can study them. You'll also learn all about the T/S 1000's *two* kinds of graphics features, and, just for fun, you'll see a complete program that will allow you to draw pictures on the TV screen.

Chapter 4—*Take Five*—is all about numbers and calculations on your computer. You'll find out how to store large amounts of numerical data in your computer conveniently and efficiently. You'll see how you can use your computer as a "super calculator." Finally, the major program in this chapter will show you how to create bar graphs from any kind of numerical information.

Chapter 5—*Words, Words, Words*—describes the features of BASIC that are designed for handling *strings*—that is, nonnumerical information. This chapter features an amusing program that may help you through those difficult moments in life when you just can't decide what to do.

Appendix A describes all the vocabulary of BASIC—that is, all the instruction words that appear on your keyboard. You will be able to use this appendix as a reference tool to help you write programs.

Appendix B summarizes the T/S 1000 error codes.

Your Timex/Sinclair 1000 and ZX81 is organized so that the information presented in one chapter is used in the next; each chapter builds on the previous chapters. To use this book successfully, you should work through it with your computer at hand, trying each new instruction as it is presented, entering and running all the programs as you encounter them. If you take as much time as you need to master each step before you go on to the next, you will finish this book with a thorough understanding of your computer's capabilities and the best ways to use them.

The most important thing you bring home from the store along with your Timex Sinclair 1000 is the tremendous enthusiasm of a new computer owner. Don't lose that enthusiasm. Let this book teach you what you need to know to make the most of your new computer.

Note to ZX81 users:

You too can use this book as a hands-on introduction to your new computer. The ZX81 offers the same programming commands, has the same keyboard, and uses the same added equipment as the T/S 1000. The only significant difference between the two computers is in memory capacity. While the T/S 1000 starts out with 2K of memory, the ZX81 has only 1K. A few of the longer programs in this book do require 2K of memory; however, if you decide to upgrade your ZX81 with the optional 16K memory module, you will be able to run even these longer programs on your computer.

Your Timex Sinclair 1000 and ZX81

Chapter 1

The Cast of Characters

INTRODUCTION

As you probably already know, the Timex Sinclair 1000 computer—amazing and powerful a machine though it certainly is, housed in its sleek little black plastic box—is of little use to you all by itself. To use it effectively, you have to attach other equipment to it. Some of these attachments are ordinary household items: a TV set, and a cassette tape recorder. Others are specialized items that are designed specifically to work with the T/S 1000 computer: for example, the printer, and the extra memory module. Most of these attachments are involved in two general sorts of jobs: sending information to, or receiving information from, your computer. These two jobs are referred to as *input* and *output*, respectively.

In this first chapter we will take a quick look at the computer and some of the equipment—required or optional—that can be attached to it. As we explore the jobs of the different components, we will learn how to attach them properly to the computer. We'll also discuss the different roles that you, the computer user, can play. By the end of this chapter, you'll be ready to start working with your computer.

THE LEADING ROLE—THE T/S 1000

Take a look at the computer unit. It's worth spending some time examining it and making note of several aspects of its design before you actually begin working with it. Figures 1.1 to 1.3 show the computer from three different angles; let's consider each one in turn.

At the front of the computer is the keyboard area. You will give commands and supply data to the computer via this keyboard. Thus, the keyboard is the computer's main input device. At first you may have trouble believing that a flat piece of plastic can really work as a keyboard. But you will soon find out that the key areas are *touch-sensitive.* This means that once the computer is turned on, a slight pressure from your finger on one of the keys will cause a message to be sent from the keyboard to the computer.

The 26 letters of the alphabet, and the 10 digits, arranged in the same order as on a typewriter, dominate the keyboard. But along with the letters and digits, the keyboard also displays an assortment of words, symbols, and tiny graphic blocks. The words are the commands and instructions that you can give the computer in order to request some action. You'll be learning what each of these words means as you read this book and as you begin experimenting with your computer.

The reason for including these command words directly on the keyboard is simple. Instead of having to type commands letter by letter, you need only press a single key to send a command or instruction word to the computer. This may not seem especially

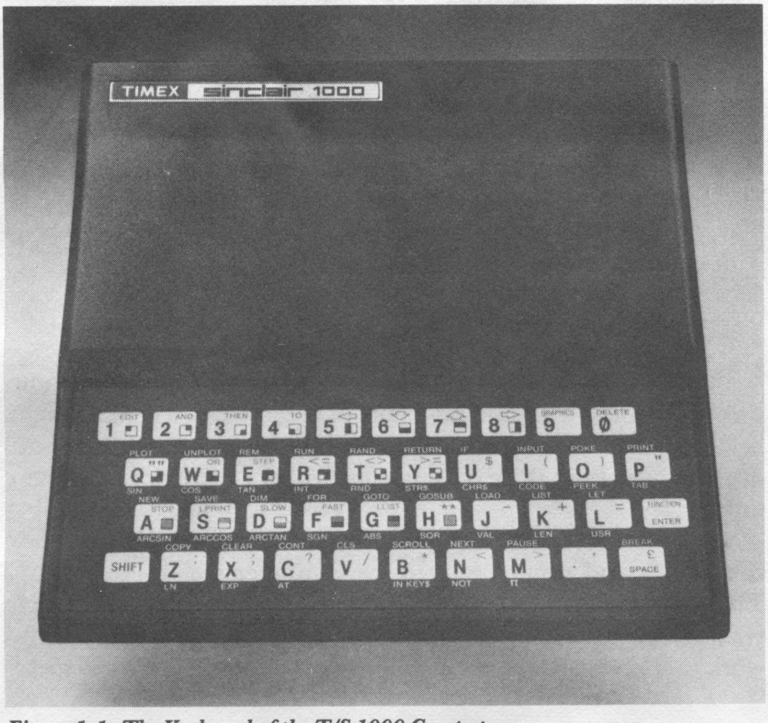

TIMEX sinclair 1000
EDIT AND THEN TO GRAPHICS DELETE
1 2 3 4 5 6 7 8 9 0
PLOT UNPLOT REM RUN RAND RETURN IF INPUT POKE PRINT
Q W E R T Y U I O P
SIN COS TAN INT RND STR$ CHR$ CODE PEEK TAB
NEW SAVE DIM FOR GOTO GOSUB LOAD LIST LET
STOP LPRINT SLOW FAST LLIST
A S D F G H J K L FUNCTION ENTER
ARCSIN ARCCOS ARCTAN SGN ABS SQR VAL LEN USR
COPY CLEAR CONT CLS SCROLL NEXT PAUSE BREAK
SHIFT Z X C V B N M . SPACE
LN EXP AT IN KEY$ NOT π

Figure 1.1: The Keyboard of the T/S 1000 Computer

convenient to you at first; starting out you may spend more time searching for the command words than you would have spent typing them letter by letter. But with practice you'll become familiar with the keyboard arrangement, and your fingers will move automatically to the keys you want. Of course, you can't perform anything like two-handed touch-typing on this keyboard, but you'll be surprised at how efficiently you'll be able to type on it after just a few days of use.

Since some of the keys have as many as five different words or symbols associated with them, a system is required for "shifting" the keyboard usage from one kind of symbol or word to another. There are several different kinds of shifting for the T/S 1000 keyboard, and we'll master them all in Chapter 2. Like the positions of the command words, the shifting methods will become second nature to you in no time at all.

Now look at the left side of your computer (shown in Figure 1.2), where you'll find four jacks. The function of each jack is clearly labeled: the first one is for connecting your TV set to the computer; the middle two are for a tape recorder; and the last one, labeled "9V DC", is for the power supply.

Included with your computer were all the plugs, cords, and other hardware that you'll need in order to make all these connections. Let's take a quick inventory of these items. One cord is about

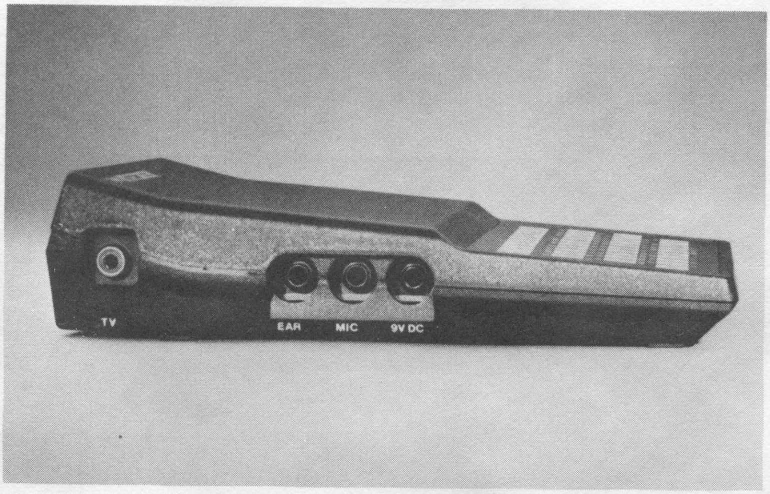


Figure 1.2: The Left View of the Computer

a yard long and has one large plug on each end. This is for the TV connection. You can go ahead and plug one of the ends (either one) into the computer jack labeled "TV" now. We'll see shortly how to attach the other end to the television set. By the way, none of the plugs that attach to your computer are dangerous for you to touch, even when the power is on.

You should also have a shorter cord with a pair of plugs on each end. This is for the tape recorder. Put it aside for now (but don't lose it); we'll examine the tape recorder connection in detail in Chapter 2.

Finally, you'll find a third cord with an AC adapter on one end and a plug on the other. The adapter plugs into any 120 volt outlet on your wall (or into an extension cord, if it's more convenient), and the small plug fits into the "9V DC" jack on the side of your computer. You can plug it in now if you want to try it out. When you do, the computer will be on and ready to operate. (Unfortunately, you'll have no idea what the computer is doing until you attach a TV.) It's a good idea to unplug the adapter whenever your computer will be unused for a long period of time.

At the back of your computer (shown in Figure 1.3) is an opening in the plastic, about two inches long and half an inch wide. This opening exposes the edge of a thin board on which you'll attach the extra memory unit or the printer, if you decide to buy either of

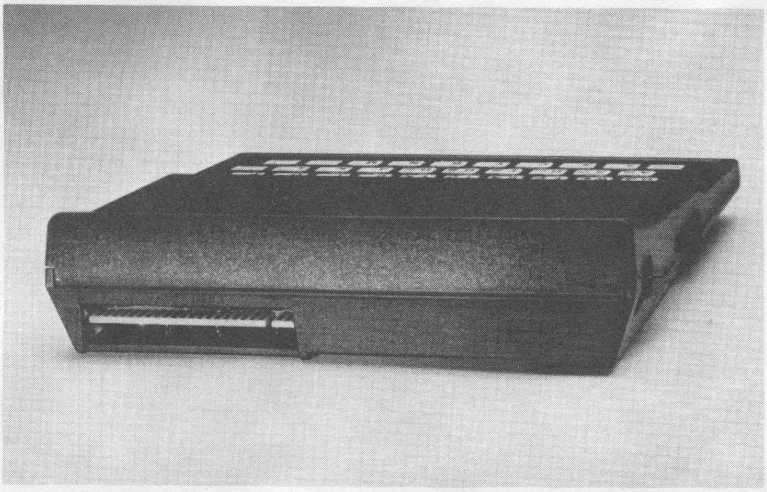


Figure 1.3: The Back of the Computer

these two optional pieces of equipment. We'll discuss them both later in this chapter. (If you already have one or both of them, a word of caution: Do not plug them in or detach them while the computer's power is on.)

Now that we have examined the outside of the computer, what about the inside—the *contents* of the black box. How much do you need to know about the internal design of the T/S 1000 in order to become a successful computer user? Undoubtedly you've begun hearing and reading about the advances in integrated circuit ("chip") technology responsible for the personal computer revolution. You might also know that the chip that "does the thinking" for your T/S 1000 is called a Z80 microprocessor. Just how much of this information do you need to master before you will be able to play a game of computer chess, or use the computer to develop your family budget?

The answer, in case you haven't already guessed, is *none.* Everything you need to know appears on the outside: the command set printed on the keyboard, and the information flashing across your TV screen. As far as you need to be concerned, the internal architecture of your computer could be summarized as whimsically as Ian Frazier did in the following passage from his short story, "The Killion":

> People unfamiliar with computers sometimes find it helpful to think of them as fairly good-sized, complicated things. Computers range in size from as small as a motel ice bucket to as large as an entertainment complex like New Jersey's Meadowlands, including the parking lot. Inside, a computer will have a short red wire hooked to a terminal at one end and to another terminal at the other end. Then there will be a blue wire also hooked to terminals at either end, and then a green wire, and then a yellow wire, then an orange wire, then a pink wire, and so on.*

Or you can believe the story about the team of highly trained fleas . . . The point of all this nonsense is this: Your new computer is designed so that its internal structure and workings need not concern you. (Notice that the black plastic box that holds your computer is not meant to be opened.) Like many other appliances in your house—from your telephone to your garbage disposal—you

*Reprinted by permission; ©1982 Ian Frazier. Originally in *The New Yorker.*

don't have to be an expert in the technology in order to make full and successful use of the machine.

With this said, we should also note that many books and magazines are available that will tell you anything you want to know about the technology of microprocessors and computers. As you work with your new computer and discover more and more about how to use it, you may find yourself becoming interested in these subjects, and you may want to read something about them. But this is not required knowledge for using your computer; the choice of what to investigate—and to what depth—is yours.

Finally, as a new computer user you'll probably be a little worried about somehow damaging your computer. Well, you can relax. There's not much you can do wrong to the T/S 1000. True, you'll occasionally read a precautionary note about one thing or another. (And then there are the common-sense precautions: don't let the dog chew on your computer; don't use the computer while you're taking a bath; don't throw it out the window of your tenth-story apartment; and so on.) But you'll enjoy your experience more if you realize that, with normal use, you really can't hurt the computer. Certainly nothing you type into the computer will ever damage any of the hardware. You may make mistakes resulting temporarily in incorrect answers or lost data; but once you've recognized such mistakes, chances are you'll be able to correct them. Mistakes are part of the learning process; the computer always forgives.

We've looked at the main computer unit; let's find out what the attachments do.

THE SUPPORTING ROLES

The TV Set

You should be able to attach any television set that you have around the house to your T/S 1000. A small, portable, table-top model would be most convenient, but whatever kind you have should work.

The role of the TV is, in short, to tell you everything you need to know about the computer's activities. When you type commands on your keyboard, you will see these commands displayed on the screen before you instruct the computer to perform them. Often you'll write a series of commands that you will want to submit to the computer all at once, for sequential performance. Such a series of commands is what we call a *program.* Before you instruct the computer to perform the commands of a program, you can display the program itself, line by line, or section by section, on the screen. You can leave the program on the screen for as long as you want, so you can study it and make sure it's right. This display of a program is called a *listing.* If you find an error in your program you can *edit* the part of the program that is incorrect; the steps of the editing process will all be displayed on the screen for you. Finally, when you *run* your program (that is, when you tell the computer to perform the commands of the program) you will in most cases see the results on the TV screen. The results may be numerical, textual, or even pictorial, because the T/S 1000 has two kinds of graphic capabilities.

Of course, the computer is always in control of what is displayed on the screen. Once you get used to reading the information that the computer puts on the screen, you will begin recognizing some fairly subtle symptoms of the computer's activity. For example, you'll be able to tell from the organization of the screen information when the computer is running short of memory space. The computer will also display screen messages—which you'll have to learn how to interpret—telling you when you've made a mistake. In most cases you'll know exactly where the mistake is and why the computer didn't like it.

If the keyboard is your means of communicating with the computer, the TV screen is the computer's most important and complete means of giving information back to you. At first you may find that there's too much information on the screen for you to absorb, but you will quickly get used to understanding the more subtle signs and symptoms displayed on the screen, and you will be able to concentrate your attention on the information that the computer is giving you.

In addition, when you write a program, you will have a good deal to say about where and how the computer places information on the screen. Several of the commands you can give the computer are directly involved with the appearance and position of numbers, text, and graphics on the screen.

Your computer connects to the television set via the small silver-colored box that came with the computer, the RF modulator (or "switch box"). Leading out of one side of the box is a wire that ends in a pair of U-shaped connectors. These attach to the TV. Look at the back of your TV set and locate the two pairs of screws labeled UHF and VHF, respectively. For the T/S 1000, you'll want to connect the RF modulator to VHF. Loosening the two screws, slip the U-shaped connectors beneath them, and then tighten them again.

On the face of the RF modulator is a switch that slides up and down. The upper position is labeled "TV", and the lower will be labeled either "computer" or "game". When you want to use your T/S 1000, you'll have to make sure the switch is in the lower position. The upper position allows you to watch TV without detaching the RF modulator.

Finally, on one end of the RF modulator is a jack similar to the TV jack on the side of your computer. This is where you plug the long cord leading from the computer. When this is done, the TV connection is complete.

You can turn your TV to either channel 2 or channel 3. On the underside of your computer you'll find a switch that must be set to correspond to the TV channel you have chosen. In general, it's probably best to use the channel that is not used for TV broadcasting in your area, but you can try both channels and choose the one that seems to give you the clearer picture.

Now switch on the TV, with the volume control turned all the way down. Plug in the adapter that supplies the power to the computer. Insert the small plug on the other end of the adapter cord into the "9V DC" jack on your computer. If all the connections are made correctly you should see a small K in "reverse video" (i.e., white on black) at the lower-left corner of the screen, as shown in Figure 1.4. This K, which stands for "keyword," tells you that the computer is ready to accept commands. You can adjust the brightness and contrast controls on your TV to get the best picture possible. (These settings will not always be the same as you like to place them for watching television.) If you're not satistifed with the picture, even after adjustments, try the other channel to see if it is better.

If you're not getting the picture at all, then check over all the connections:

- The computer's power adapter should be plugged into the wall outlet; the small plug on the other end of the cord should be inserted in the "9V DC" jack at the side of your computer.

- The U-shaped connectors of the switch box must be attached to the VHF (not the UHF) screws at the back of your TV set, and the switch should be set for computer use. The long cord should be plugged securely into both the switch box and the TV jack on the side of the computer.
- The TV channel you are tuned to (either 2 or 3) must correspond to the setting of the switch located on the underside of the computer.

With the TV attached to your working computer, you are ready to start writing commands and experimenting with the power of the T/S 1000. In fact, you may well decide not to connect any other equipment to the computer at the beginning. You can learn a lot about your computer with nothing other than a TV set to show you what the computer is doing.

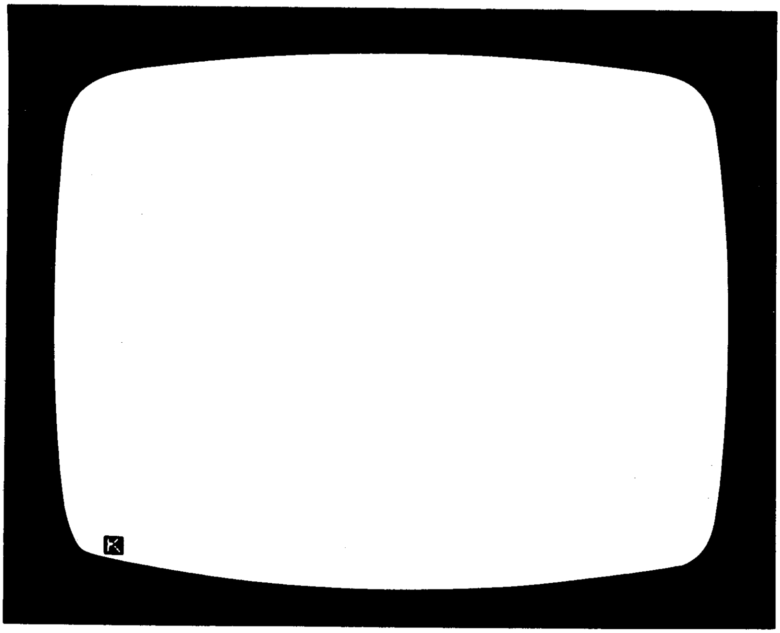


Figure 1.4: Turning on the Computer: The K Cue

Eventually, however, as you grow into the computer experience, you will probably want to add other capabilities to your system. You may want to add extra memory to the computer, in order to write longer programs. You may also want to start saving your programs in some permanent form—or you may want a convenient way of running other people's programs on your own computer. Or, finally, you might want to record some results from the computer onto paper. The following sections of this chapter describe the different pieces of equipment that allow you to do all these things.

Extra Memory

The computer has two kinds of memory; one kind is permanent and the other is temporary. You may already be familiar with the names of these two kinds of memory: ROM (for Read Only Memory) and RAM (for Random Access Memory), respectively. ROM, the permanent memory, stores all the information that defines the "personality" of the T/S 1000. This information determines what commands you can give your computer, and how the computer will interpret and carry out your commands. The information in ROM is never lost, even when you turn off the power of your computer.

The other kind of memory, RAM, stores information about the current activities of the computer. For example, it holds any commands or any program that you have typed into the computer. It keeps track of the images that the computer is sending to the television screen. It also holds data that you have submitted to the computer, and stores the intermediate and final results of calculations you tell the computer to perform. RAM, then, is very dynamic; the information it stores can change from moment to moment. The most important thing to remember about RAM is this: When you unplug the power, all the information stored in RAM is lost.

The computer's memory is measured in units called *bytes*. A byte is the amount of memory that the computer requires to store the information from one keystroke on the keyboard. So if you type any of the commands or characters printed on the keyboard, the computer needs one byte to keep track of it.

Now, your T/S 1000 has about 2000 bytes of RAM built into it. (Actually, we say it has 2K bytes, where the K means 1,024.) You will see that this is enough memory for you to write short, but interesting and useful, programs for the computer to perform. It is plenty of memory for you to work with while you are just learning how to use your computer.

At some point, however, you may begin to wish your computer had more memory. You may want to write longer programs yourself, or run very long programs that other people have written. If you reach this point, you can buy an extra memory module to attach to your computer. Timex produces a 16K RAM module, and other companies produce modules that will give you up to 64K. How much memory you need will depend on what you finally decide you want to use your computer for.

These modules slide into the exposed piece of board at the back of your computer. If you've already bought a memory module, you can attach it right away. (Again, always be careful to unplug the power before you attach or disconnect a memory module.) All the programs we will write in this book will fit in 2K of memory; however, we will discuss some programs you can buy that require 16K.

If you haven't yet bought an extra memory module, you should probably wait a while before you decide to buy one. They will always be available later, and, as we have already mentioned, you can get a thorough introduction to the capabilities of your computer simply with the built-in 2K RAM.

A Cassette Tape Recorder

Remember that we defined a *program* as a sequence of instructions that you submit all at once to your computer. When you *run* a program, you instruct the computer to perform the instructions, one by one, in some specified sequence. The other point to remember about programs you write is this: they are stored in RAM, the temporary memory. This means that if you work on a program and then disconnect the computer from its power supply—accidentally or on purpose—the program you've been working on will be lost.

But your computer provides a nice solution to this problem. You can store programs by recording them directly from the computer onto a cassette tape. Then, later, when you want to run a program again, you can play it back, and the computer will store it once again in RAM. To be sure, this process requires some rather careful connections between the tape recorder and the computer;

we will discuss these connections in Chapter 2. But recording computer programs does not require expensive or sophisticated recording equipment. If you have any cassette tape recorder around the house, the chances are good that it will work with your computer. If you don't have one, you may eventually want to invest in one, but you should use an inexpensive one.

The Printer

One other piece of equipment that you might consider buying is the printer designed to be attached to the T/S 1000. Like the extra memory module, the printer connects to the exposed piece of board at the back of your computer, and you should unplug the computer before you attach it. The printer uses rolls of special paper; the documents it produces, while readable, are only of medium quality. All the same, the ability to produce a printed record from your computer may often be a valuable asset.

The computer has several commands that allow you to control the printer directly from the keyboard. You can, for example, instruct the computer to list a program on the printer, or to copy the entire contents of the TV screen onto paper. You can also write programs that will send their results to the printer rather than to the screen. If you decide to buy a printer, you will find that it's extremely easy to use for producing all kind of records from your computer. (Appendix A describes the commands that you'll use to work the printer: COPY, LLIST, and LPRINT.)

THE PRODUCER, DIRECTOR, AND WRITER—YOU

Next, we turn our attention to *you,* and the different roles you can play in making your computer work for you.

We have seen that your computer is designed to accept and perform certain commands, which you can communicate to it by pressing the appropriate keys on the keyboard. If you look at the keyboard you'll see that there are five dozen or so different words that the computer understands. You can think of these words—along with the specific rules for using them—as a kind of language: a *computer* language. This computer language is much simpler than a human language; it has a very limited vocabulary, and a simple, regular grammar. For this reason, you'll have a much easier time learning it than you would a foreign language like French or Italian.

In addition, most of the words in its vocabulary are familiar words in English. They will be easy for you to remember once you have learned their new meanings in the context of telling the computer what to do.

There are many different computer languages. The one that is built into the T/S 1000—and, in fact, into most small computers—is called BASIC (an acronym for Beginner's All-purpose Symbolic Instruction Code). You'll be hearing more and more about BASIC in the next few years. Because it is such an easy language, people everywhere are learning how to use it to solve all kinds of problems on computers. Even small children are beginning to learn to write programs in BASIC in elementary school. Soon, BASIC programming may be as common a home and office skill, as, say, typing or using a pocket calculator.

BASIC is growing in popularity because it can be used by people who are not necessarily computer experts. Yet you'll be surprised to find out how powerful BASIC is when you use it on your computer. This simple language actually has much in common with other languages that professional computer programmers use on much larger computers than the T/S 1000.

The first role you will play with your new computer, then, will be as a BASIC programmer. When you think of a job that you want your computer to do for you, you'll find yourself automatically organizing the job in the following way:

1. You'll divide the job into a series of elementary steps that the computer can perform one after the other. These steps may include calculations, repetitive tasks, or even decisions that you'll want the computer to make based on available data.
2. You'll determine the most efficient way to express these steps as a series of BASIC instructions. If your program will be a long one with many instructions, then you'll probably start out writing it down on paper rather than typing it into the computer extemporaneously.
3. Once you have developed a program, you'll type each line into the computer. Then you'll put the program through a trial run to see if it really accomplishes the job you had planned for it. If the program performs calculations on numerical data, you may try it out on several different sets of trial data before you're satisfied that it is correct.

4. If the results of your program are incorrect, you'll return once again to the BASIC instructions you have written, and you'll try to figure out what's wrong. Often the nature of the incorrect results will tell you right away what the problem is; at other times the errors will be more subtle and you may ponder for hours—or even days—before you solve the puzzle of your own program!

You'll see your first BASIC program in Chapter 2. Actually, with this first program you'll be concentrating more on using the T/S 1000 keyboard than on the meaning of the BASIC instructions, but very soon you'll be writing your own programs. Notice that the process of developing a program, in the four steps outlined above, includes a course of action for detecting and correcting mistakes. It is almost inevitable that new programs will start out with errors in them. This is nothing to be embarrassed about. The T/S 1000 will catch some errors you make as you are writing your program. Other errors will appear when you run the program. We will see that the computer has several ways of helping you to find and correct errors in your programs.

When you bought your computer, you probably noticed that you could also buy some programs—written by other people—that were designed for use on the T/S 1000. These programs are stored on cassettes, and all of them are fairly expensive. But this is another way you can use your computer—by running other people's programs on it to perform specific tasks. In Chapter 2 we will discuss some of the advantages and disadvantages of purchasing programs (as opposed to writing them yourself). For now, you may be asking yourself the following question: Why should you go through the trouble of learning to program in BASIC if programs have already been written that you can simply buy and run?

In fact, many people who buy the T/S 1000 computer may never learn to write a BASIC program. They will always select and purchase programs from their computer dealers, according to their specific needs. This is a legitimate use of the computer; you don't *have* to learn to program in order to use your computer successfully. However, if you never attempt to learn BASIC and write your own programs, you'll be missing out on a significant opportunity for self-education. If you learn to design your own programs, your computer will become a valuable tool that you can always apply to your own individual computing needs. In addition, the process of learning to program a computer will give you new insight into an instrument that is becoming increasingly important in our society.

SUMMARY

To communicate with the computer, we need to attach input and output equipment. On the T/S 1000, the major input device is the built-in keyboard. The keyboard conveniently displays all the commands that you can give your computer, so that each word can be entered by pressing a single key.

The TV screen is the major output device, and it supplies a great deal of information about your programs, their results, and the current action of the computer. In addition, you can attach a printer to your computer if you want more permanent output records.

A simple cassette tape recorder can also be connected to the T/S 1000 when you want to save programs that you have written, or when you want to run programs that you buy. Finally, extra memory modules are available to increase the temporary memory capacity (RAM) of your computer.

The words that you see on your keyboard make up the vocabulary of the BASIC language. Learning to program in BASIC, while not absolutely essential for computer use, is likely to make your computer experience more efficient and more satisfying.

Chapter 2

The First Act: Enter Your Program

INTRODUCTION

If you've followed through the instructions detailed in Chapter 1, then your computer is on, the TV set is attached, and you're ready to start to work. The first task ahead of you is learning to use the computer's keyboard; that's the main goal of this chapter. We'll begin by taking a slightly closer look at BASIC. We'll see what a program looks like, and we'll discuss, in general terms, the way a program is organized. Then you'll enter a short program into your computer. Along the way, you'll see how the computer helps you identify and correct any errors you might make.

Next, you'll see how to use a cassette tape recorder with your computer. We'll look in careful detail at the connections you have to make in order to save programs on a cassette and to load a program back into the computer from a cassette. You'll try out the process by saving the program that you will have typed into the computer.

This will lead us to a third topic: programs you can buy on cassette. We'll look at an example, and then we'll end the chapter with a consumer's guide to buying programs for your T/S 1000.

THE COMPUTER LEARNS ITS LINES

Every good computer must, of course, learn to be friendly, so the first program we'll look at will guide the computer through the following tasks:

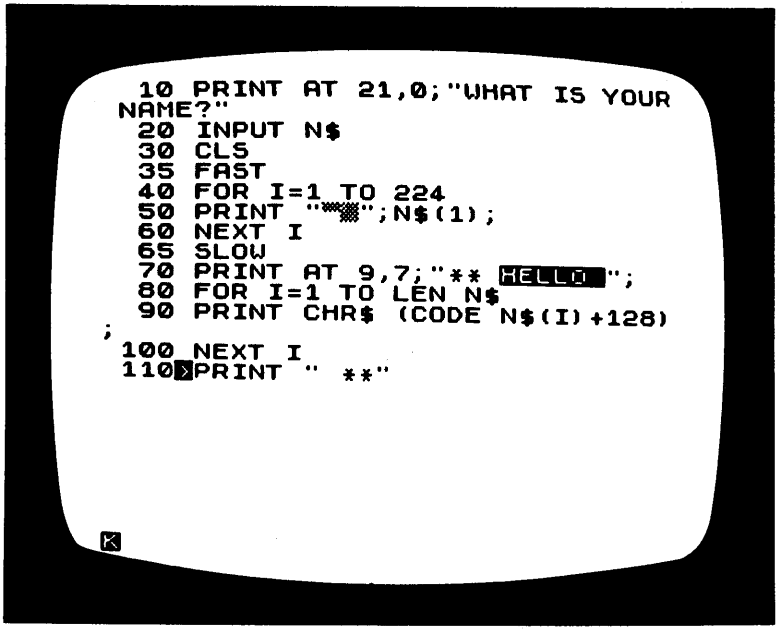
1. ask you your name;
2. draw a design on the TV screen;
3. print a greeting on the screen.

You may consider these somewhat frivolous jobs for a computer program; but at the moment we're not worried about usefulness. The purpose of this program is to give you a thorough introduction to using the keyboard. By the time you've typed in the program, you will have seen examples of everything the keyboard can do for you.

The BASIC Computer Language

The "greeting" program is shown in Figure 2.1. Take a moment to study it before you begin typing it into your computer. You can learn a lot about the BASIC language just by looking at a BASIC program.

We learned in Chapter 1 that BASIC has a very limited vocabulary, mostly made up of familiar words in English. Indeed, the greeting program has only a few words that seem to be written in some kind of abbreviated shorthand (CLS, LEN, CHR$); most of the other words are familiar enough. This doesn't mean that you



```
  10 PRINT AT 21,0;"WHAT IS YOUR
 NAME?"
  20 INPUT N$
  30 CLS
  35 FAST
  40 FOR I=1 TO 224
  50 PRINT "▒";N$(1);
  60 NEXT I
  65 SLOW
  70 PRINT AT 9,7;"** HELLO ";
  80 FOR I=1 TO LEN N$
  90 PRINT CHR$ (CODE N$(I)+128)
;
 100 NEXT I
 110 PRINT "  **"
```

Figure 2.1: The Greeting Program

can immediately understand what each line of the program will do; but at least you can see that the program does not, after all, appear very intimidating.

What does this program tell us, then, about the characteristics of BASIC? First, a BASIC program is organized into lines, and each line is numbered. This program has 13 lines. The length of a program can vary from a single line to hundreds of lines, depending on the task it is designed to accomplish.

The lines in this program are numbered from 10 to 110, but notice that the intervals are not regular. Most of the line numbers increase by increments of 10, but some increase by 5. When you write a BASIC program, you are free to number the lines in any way you want. To the computer, the only significance of the line numbers is that they indicate the order in which the lines should be dealt with. This program could have been numbered from 1 to 13 or from 100 to 1300, and the computer would still have accepted it. The advantage of incrementing the lines regularly, say by 10, is that once you have written a program you will often want to add lines to it—second thoughts—somewhere in the middle. This is easy to do, as long as your numbering system leaves "room" for extra lines. For example, lines 35 and 65 of this program were added after the rest of the program was written.

Two of the numbered lines in this program—10 and 90—happen to take up more than one line on the TV screen. This is perfectly all right. The computer organizes the information on the screen so that each *screen line* can hold up to 32 characters, but a line in a BASIC program can be much longer than that. When a program line exceeds the number of characters available on a screen line, it is simply continued on the following screen line. This can produce some unusual cuts in the lines of a program—for example, the last character of line 90, a semicolon, had to go by itself on the next screen line—but it doesn't affect the way the computer reads and executes the line.

The first word of every line in a T/S 1000 BASIC program is always a *keyword.* The keywords tell the computer what kind of instruction is coming up. Each keyword must be followed by its own particular "grammar"; that is, the computer knows what kind of instruction patterns to expect after each keyword. There are 26 keywords on your keyboard, located above each of the letter keys. Figure 2.2 shows the location of the keywords.

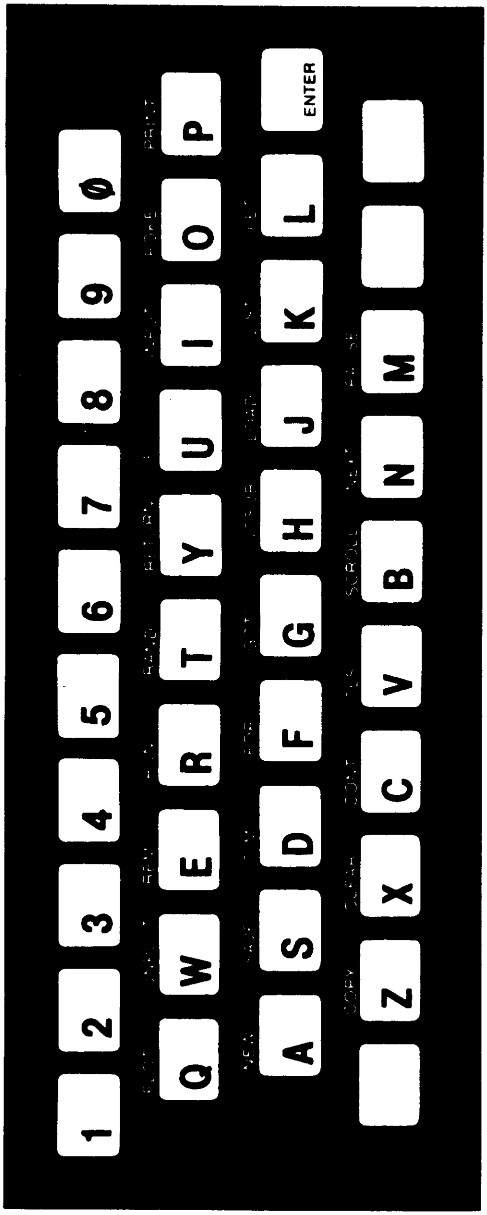


Figure 2.2: The Keywords

When you type a numbered line of a BASIC program into your computer, the computer stores the line in its memory and displays the line on the screen for you. It does *not* begin performing the instructions of your program until you tell it to do so. It merely keeps track of the program, and waits for you to tell it what to do next. We'll see how this works after we've described how to enter the program.

The Keyboard—Keywords and Modes

At the moment, your screen is blank except for the reverse-video K in the lower left corner. You'll recall that the K stands for *keyword.* This *cue* (referred to as a *cursor* in your user's manual) is the computer's way of telling you that it is ready to accept any keyword from the keyboard. When you start typing in the program, you'll see other cues—including L, G, F, and S, all in reverse video. These cues tell you what "mode" the keyboard is in—that is, what kind of words or symbols you can enter next into the computer.

We'll go over each line of this program keystroke by keystroke; you should type the lines one by one as you read the descriptions. (You'll also read a brief, general explanation of what each line *does;* but, for the moment, concentrate on the keyboard, not on the meaning of the lines. You'll have plenty of time to master BASIC later, after you've learned how to use the keyboard.)

Entering the Program, Line by Line

The first line of the program is a PRINT instruction. PRINT tells the computer to display some information on the TV screen. In this first line, PRINT is followed by the word AT, which is one of several possible ways of specifying exactly *where* on the screen you want information to be displayed. The "information" in this case is a question, WHAT IS YOUR NAME?, which appears within quotes at the end of the PRINT instruction.

So let's type the line. If you've typed anything at all on your computer yet, you will have noticed that the keys don't require very much pressure to register. Just a light touch is all that's needed. You'll have to watch the screen as you type, however, to make sure each key you've touched has in fact sent its message to the computer.

As we've seen, the first thing you have to type at the beginning of each line of your BASIC program is the line number. For the first line, type the digits 1 and 0. As you enter each character, you'll see that the K cue moves one space to the right, and the digits appear to its left.

Next, find the PRINT command. It's located above the [P] key (at the far right of the keyboard, the second row down). Touch the key and see what happens on your screen. The whole word PRINT appears at once on the line. Also, the reverse-video cue changes to an L. This tells you that you have left the keyword mode and are now in the *letter* mode; the next key you press will register as a letter or a digit, not a keyword.

To enter any keyword, then, all you have to do is press the single key on which the keyword is displayed. In fact, you *cannot* type out the letters of a keyword. If you were to type the letters, P-R-I-N-T, the computer would *not* recognize it as the keyword PRINT. Whenever entering a keyword is correct in the special grammar of BASIC, the computer tells you by displaying the [K] cue. If you don't see the reverse video K, you know that you can't enter a keyword.

The next word in the first line is AT. AT is one of 25 *functions* available on your keyboard. The functions appear beneath all of the letter keys (except the [V] key). They are shown in Figure 2.3. Some of these functions are genuine mathematical functions, such as sine (SIN), cosine (COS), and the natural logarithm (LN). If you intend to use your computer for scientific or mathematical applications, you undoubtedly already know about these functions. Other functions on your keyboard are not mathematical at all. You might think of them as programming functions, because once you know how to use them, they will simplify many programming tasks. AT, for example, is a "function" that allows you to specify, very conveniently, the location of information placed on the screen.

All the functions—mathematical or otherwise—have one thing in common. To type a function, you have to "shift" the keyboard into the function mode. To perform this shift, you press two keys at once:

- the SHIFT key, located at the lower left corner of the keyboard, and
- the FUNCTION/ENTER key, located at the right of the keyboard, just below the [P] key.

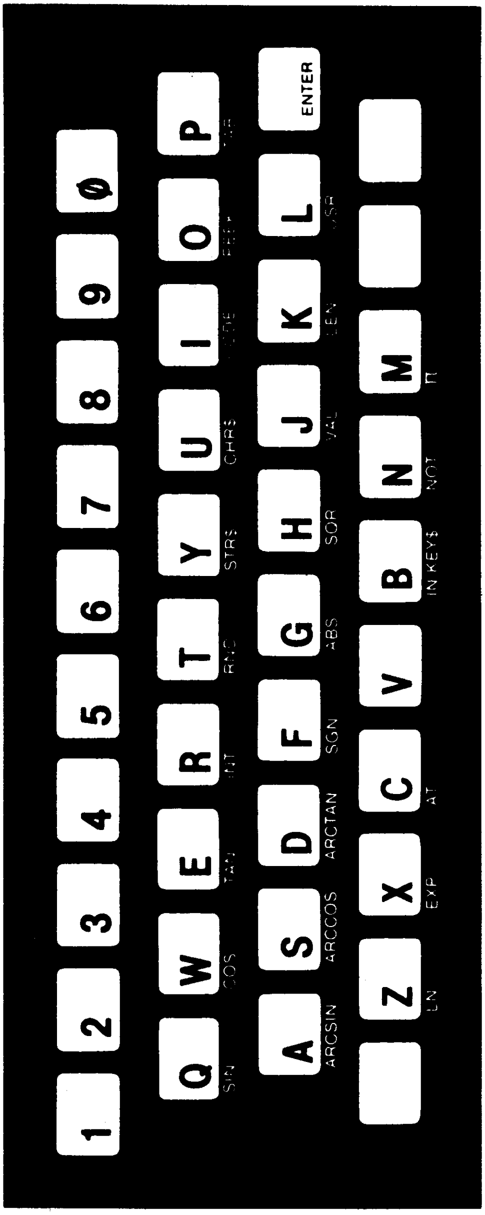


Figure 2.3: The Functions

First hold one finger down on the SHIFT key, then touch the FUNCTION key, and watch what happens on the screen. The reverse-video cue changes from an L to an F. You are now in the function mode. You can press any of the letter keys that have functions below them, and the function will appear on the screen as part of your BASIC line.

Locate the AT function on your keyboard. It's below the [C] key. Press the key, and AT becomes part of your line. Your line now reads:

```
10 PRINT AT
```

Notice that the reverse-video cue has once again changed back to an L.

Like the keywords, the functions must be entered as single keystrokes. Typing the letters A-T in the letter mode would not mean the same thing at all to the computer as typing the AT function in the function mode.

Next we have two numbers to type onto our line. As you may have guessed, these numbers represent the "address" on the screen where the computer will print the message. (The numbers 21,0 mean row 21, column 0; we will see exactly how this works when we study the AT function, in Chapter 3.)

Before we go any further with the typing, we should find out what to do in the event of a typing error. Such errors are inevitable; perhaps you have made one already, and are a little confused about what to do next.

The computer gives you a very easy way to "erase" anything you type on a line. Again, it involves pressing two keys at once:

- the SHIFT key, and
- the [0] (zero) key.

You should notice two things about the [0] key, at the upper right corner of the keyboard. First, to distinguish the digit 0 from the letter O, the zero has a slash through it. (The slash also appears on the screen.) Second, the zero key has the word DELETE written, in red, above the digit. So, pressing SHIFT and DELETE at the same time, you can "erase" any mistakes that you might make while you are typing.

Try it once now. Type some letter that doesn't belong on your line. Then press the SHIFT key down with one finger and touch the DELETE key with another. Notice what happens on the screen. The L cue backs up over the unwanted letter. The letter is gone, and you can continue on with your typing.

Now type the two numbers, starting with 21. The 21 is followed by a comma (,) and the 0 by a semicolon (;). This punctuation, though it may not make immediate sense to you, is an essential part of the meaning of the line, so it has to be typed correctly. The comma is located in the lower right area of the keyboard, on the same key as the period (.). The semicolon is located on the [X] key. Both the comma and the semicolon appear in red on your keyboard.

In fact, every key on the keyboard has a character or word that is printed in red. We will refer to these as the "shift" characters. To type them, you must first hold the SHIFT key down, and then, with another finger, touch the appropriate key. All of the shift characters are shown in Figure 2.4.

Try typing the comma. If you should get a period at first instead of a comma, it means that you're not pressing the shift key quite hard enough (or in exactly the right place). Delete the period (SHIFT-DELETE) and try again. Then type the zero and the semicolon, to complete the address. The line on your screen should now be:

```
10 PRINT AT 21,0;
```

The rest of the line is the actual message that will be printed on the screen when the computer performs this PRINT instruction. The message is between quotation marks. The quote character is on the [P] key; it is a shift character. Press SHIFT-P; the quotation mark should appear on the screen. Now type the message: WHAT IS YOUR NAME, letter by letter. (The space key is in the lower right corner of the keyboard.) You'll notice when you type the N of NAME that you have run out of room on the screen line. When you type the A, the computer automatically moves you down to the next line. The message ends in a question mark, which is a shift character located on the [C] key.

Before you type the ending quotation mark, we're going to experiment with another special feature of your computer. Normally, you would finish typing the line, and then press the ENTER key to tell the computer that the line is complete. Now, however, to see what happens when you make a mistake, press ENTER *before*

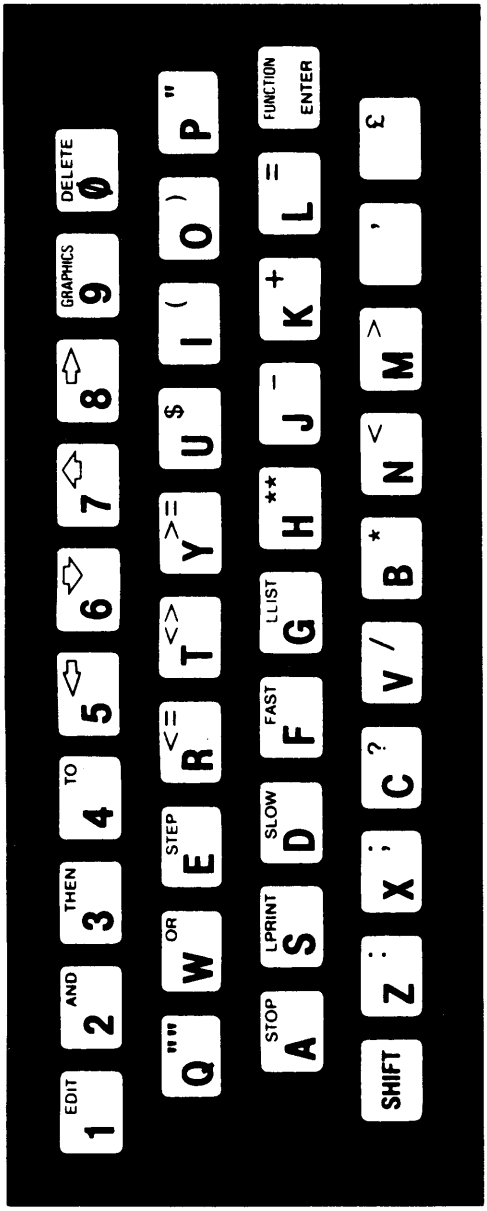


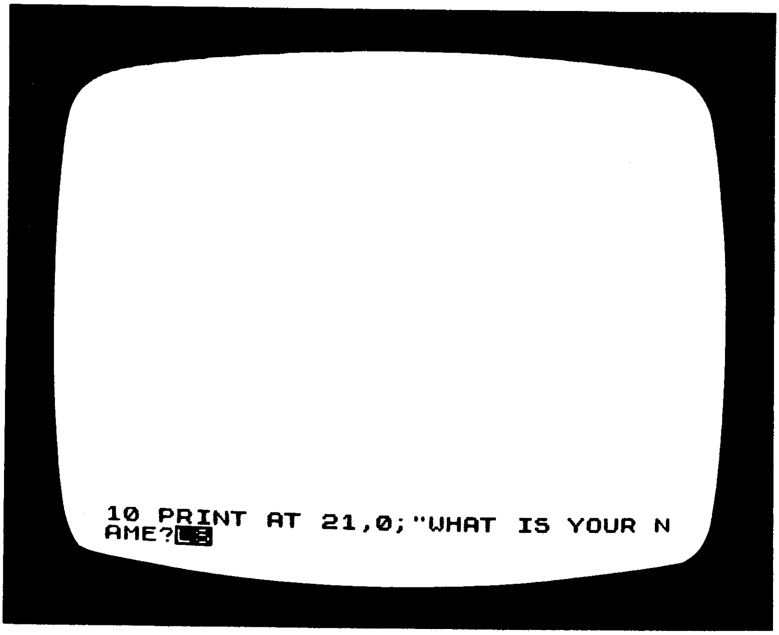
Figure 2.4: The Shift Characters

you type the last quotation mark.

Automatic Error Detection

What you'll see on the screen after you press ENTER is shown in Figure 2.5. A new cue has appeared next to the L cue. The S cue stands for *syntax error.* The word *syntax* refers to the rules of the special "grammar" that you must always follow when you write a BASIC program. When you write a line that breaks one of these grammar rules, the computer notices it right away, and refuses to accept the line as part of a program. The S cue is the computer's gentle way of telling you that you've done something wrong; you have to take another look at the line in question.

Of course, since you made this particular mistake intentionally, you know exactly what is wrong. The final quotation mark is missing. Now, when you type the quotation mark (SHIFT-P), the S cue disappears, and your line is complete. Press the ENTER key again and watch what happens. Figure 2.6 shows the results. The correct program line moves to the top of the screen. This means



```
10 PRINT AT 21,0;"WHAT IS YOUR N
AME?LS
```

Figure 2.5: The computer responds to a syntax error.

that the computer has stored it away in its memory as the first (and currently, the only) line of a program. The K cue has reappeared in the lower left corner of the screen, telling you that the computer is now ready for your next instructions.

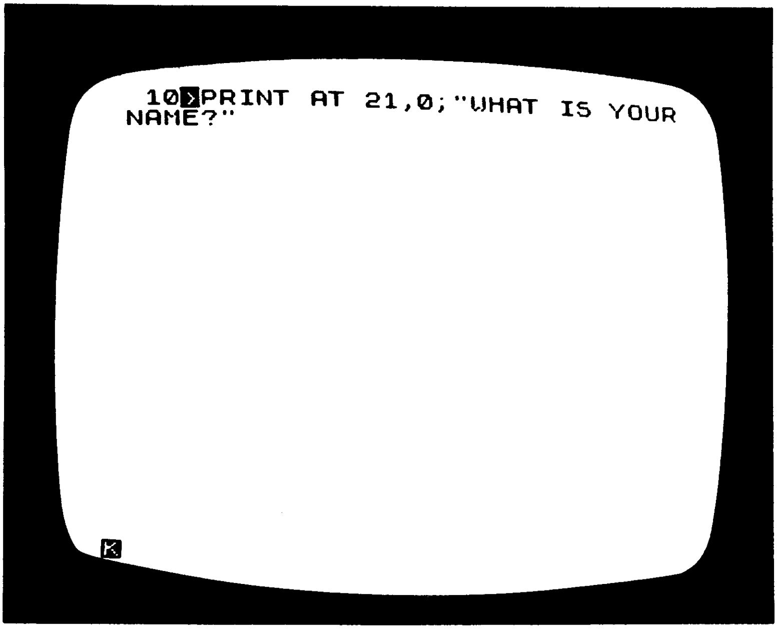
The Second Line: Using the Editor

Now let's type the second line of the program, line 20:

```
20 INPUT N$
```

The keyword INPUT tells the computer to interrupt the performance of a program, temporarily, and wait for you to type some information on the keyboard. Using the INPUT command in your program, you can create a kind of *dialogue* between the computer and the person using the computer. You'll see exactly how this process works, a little later, when you finish typing this program and then run it.

The characters N$ after the keyword INPUT represent what we call a *variable name.* Don't let this terminology intimidate you. A



```
 10>PRINT AT 21,0;"WHAT IS YOUR
NAME?"
```

```
K
```

Figure 2.6: Entering a Program Line Correctly

variable is simply a place that the computer reserves in its memory for a specified type of information. You can define many variables for use in a program, but you must give a name to each one; the computer can then keep track of the variables by name. The data contained in a variable can change many times during the course of a program—that's why it's called a variable—but the *name* never changes.

When the computer performs the instruction INPUT N$, it first waits for some kind of data to be input from the keyboard. When the data arrives, the computer stores it in a variable that it gives the name N$. Henceforth, whenever you refer to the name N$, the computer will know that you are interested in the data stored in that variable.

You can define two types of variables for use in a program—variables destined to store numerical data, and variables for textual (nonnumerical) data. In computer parlance, we call a nonnumerical data item a *string.* To define a string variable, you have to specify *in the name of the variable* that you want the computer to reserve memory space for a string. You do this by adding a dollar sign ($) to the end of the variable name.

So, once again, INPUT N$ tells the computer to accept a string from the keyboard and to store that string some place in its memory under the name N$. If the concept of a variable still seems a bit hazy to you, don't worry too much about it at this point. We will review it again later on. Remember that our primary goal right now is to learn to use the keyboard. The reason for discussing variable names at this point is actually to help you understand another error-checking feature of your computer—the editor.

When you type line 20 into your computer, type it, at first, as follows, with the intentional omission of the "$" character:

```
20 INPUT N
```

Grammatically, this is a perfectly good instruction. It says, "accept some *numerical* data from the keyboard, and store it in memory under the name N." *Numerical* data, because the character "$", which indicates a string variable, is missing from the variable name. Enter the line this way for now. The keyword INPUT, by the way, is located above the [I] key. Type the line, and press the ENTER key. The line will jump to the top of the screen, just below line 10.

Often while you're writing a program, you'll make a mistake like this. You've entered a perfectly good line into your program, but it's just not the line you wanted. Since the line is a good one grammatically, the computer has no way of knowing that there's anything wrong, so it accepts the line exactly as it is written. Once the line is entered, and you examine the program on the screen, you will probably recognize the error and want to change it. The T/S 1000 computer supplies an extremely simple way to correct such errors. This feature is called the *editor.*

Since the line that you want to edit is the line that you entered most recently, you can use the editor directly. Find the word EDIT on your keyboard; it is located on the [1] key. EDIT is printed in red, so you know that it requires a shift. Hold the SHIFT key down, and then touch the EDIT key. Figure 2.7 shows what will happen on your screen.

You can see that a copy of the line has been moved back down to the bottom of your screen, the area where all new lines appear as you type them. The K cue appears inside the line, just after the

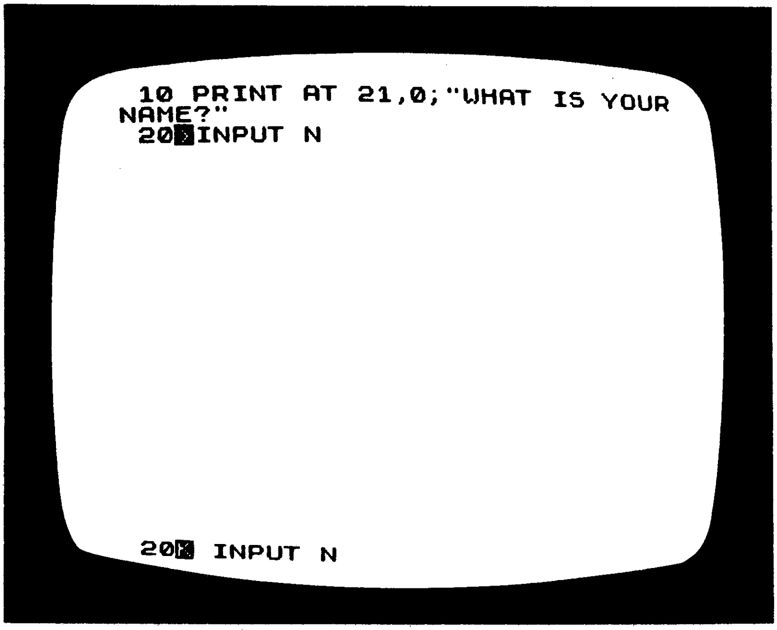

```
 10 PRINT AT 21,0;"WHAT IS YOUR
NAME?"
 20>INPUT N




 20K INPUT N
```

Figure 2.7: Editing the Most Recently Typed Line

line number. You can now use the left- and right-arrow direction keys (SHIFT-5, and SHIFT-8, respectively) to move the cue back and forth in the line. For deletions, you can move the cue to the immediate right of the character or word you want to delete, and then press SHIFT-DELETE, just as though this were a new line.

In this case you want to add a character to the line, specifically the $ character at the end of the variable name. Press SHIFT-8 (that is, the SHIFT key and the [8] key together) twice to move the cue to the correct position, just after the N. The dollar sign is also a shift character, located above the [U] key. Press SHIFT-U to add the character to the line. Then press ENTER to reenter the edited version of the line. You will see the corrected line appear at the top of your screen. And, once again, the K cue appears at the lower left corner; the computer is ready for your next command.

Up to now you've entered only two lines of the program, but in the process you've learned a lot about your keyboard and your computer. Let's review briefly what you've learned:

- When you first begin typing a line into your computer, it appears at the bottom of your screen. If you make a typing error at this point, you can "erase" the previous keystroke by pressing the SHIFT key and the DELETE key at the same time.
- The first word of every line is a keyword, which you enter in a single keystroke. Afterwards, the cue changes to a reverse-video L, for *letter mode.* If you want to enter a function, you must first shift to the *function mode* by pressing SHIFT-FUNCTION. All the functions are also entered with a single keystroke.
- All the characters and words that appear in red on your keyboard are shift characters. To type one of them, you must first hold the SHIFT key down and then press the appropriate key.
- When you have completed a line, you press ENTER to tell the computer to store the line as part of the program. If the line contains no syntactical errors, it will move up to the top of the screen, with the rest of the program listing.
- If you try to enter a line that has a syntax error, the computer will find the error and display the S cue to tell you where the

error is. You can use the left- and right-arrow keys (SHIFT-5, SHIFT-8) to move the cue around inside the line, and the DELETE key (SHIFT-0) to "erase" any errors. You can also add words or characters if necessary.

- To correct an error in a line that has already been entered into the program, you use the edit feature, which is invoked by typing SHIFT-1.

We will move more quickly through the remaining lines of the program, but you should remember to practice using the computer's error-correction features whenever you need them. When you finish, your screen should look like the program listing in Figure 2.1.

Finishing the Program

The next two lines, 30 and 35, each consist of one-word instructions. Line 30 is:

```
30 CLS
```

The keyword CLS, which is located above the [V] key, instructs the computer to clear the screen of all information. The instruction in line 35 puts the computer in the *fast-calculation* mode (which we will discuss in Chapters 3 and 4):

```
35 FAST
```

The word FAST is a shift character, located on the [F] key.

The next three lines, 40 to 60, work together to instruct the computer to repeat a certain task many times. We call the action resulting from this series of lines a *loop,* because the computer performs the lines, and then loops back to perform them again and again. The keywords that define this kind of loop are FOR (line 40) and NEXT (line 60). In this particular example, the task to be repeated is defined in a single line (line 50), but it is possible to construct a loop that repeats many lines. The end of the loop is always marked by the NEXT line. The FOR line, in addition to indicating the starting point of the loop, specifies exactly how many times the task should be repeated. We will examine the syntactical details of the FOR/NEXT loop in Chapter 3.

When you see the results of this program, it probably won't be

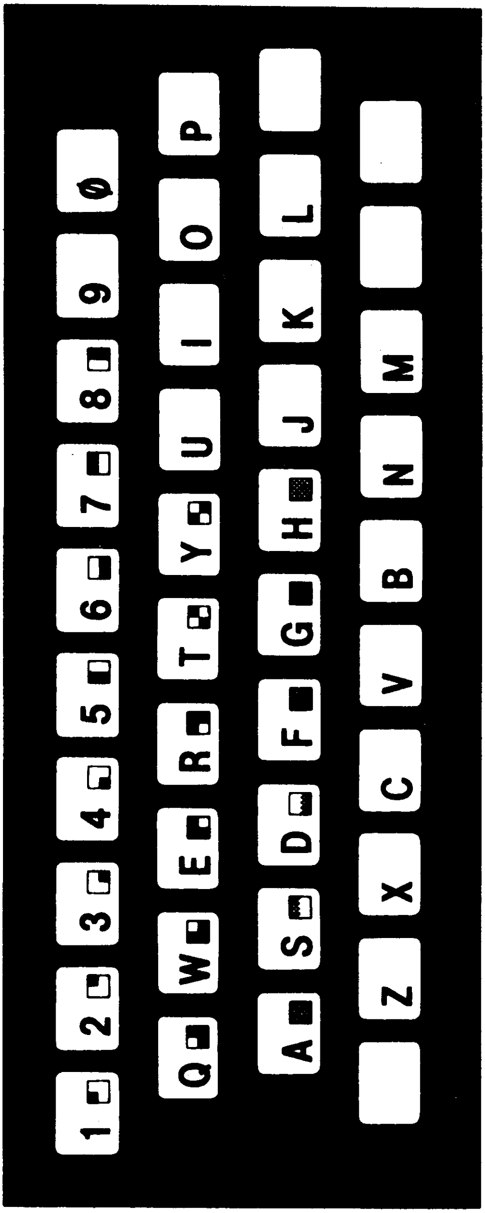


Figure 2.8: The Graphics Characters

at all difficult for you to guess what the repeated task is. Enter line 40:

```
40 FOR I = 1 TO 224
```

The keyword FOR is located above the [F] key. The equal sign (=) and the word TO are both shift characters, located on the [L] key and the [4] key, respectively.

Line 50 presents yet another new feature of the keyboard. Look back at line 50 in Figure 2.1 to see what this new feature is. The line consists of a PRINT instruction. After the keyword, PRINT, there are two *graphics characters,* between quotation marks. The T/S 1000 computer has 20 graphics characters, which can be displayed in any combination and at any location on the TV screen. On the keyboard, these characters are located on the first eight digit keys and on twelve of the letter keys, as shown in Figure 2.8.

To type these characters, you must first shift into the *graphics mode.* Begin the line by typing the line number, the keyword, and the opening quotation mark:

```
50 PRINT "
```

The L cue tells you that you are currently in the letter mode. To shift into the graphics mode, press the SHIFT key and then touch the [9] key with another finger. (Notice that the [9] key has the word GRAPHICS, in red, above the 9.) When you do this, the cue will change to a reverse-video G, telling you that you are now in the graphics mode.

All the graphics characters on your keyboard are shift characters. The two that you want for this line are located on the [S] key and on the [H] key. Start by pressing SHIFT-S. The character will appear on the line, and the computer will remain in the graphics mode. Now press SHIFT-H for the second character. Figure 2.9 shows what your screen should look like at this point.

To complete the line you now have to shift out of the graphics mode and back into the letter mode. To do this, simply press SHIFT-9 again. Just remember that the word GRAPHICS, on the [9] key, gets you both into and out of the graphics mode.

Back in the L mode, you can finish typing the line by entering the following sequence of characters:

```
";N$(1);
```

All the punctuation is important, so don't miss any of it. The 1 in parentheses after N$ refers to the *first character* of the string stored in the variable N$. (You'll recall this when you see the results of the program.) The open and close parentheses are both shift characters, located on the [I] key and the [O] key, respectively.

Line 60 completes the loop:

```
60 NEXT I
```

The keyword NEXT is located above the [N] key. Line 65 puts the computer back into the *slow-calculation* mode, which we will discuss in Chapters 3 and 4:

```
65 SLOW
```

The word SLOW is a shift character, located on the [D] key.

Line 70 once again makes use of the T/S 1000's special graphics capabilities. The graphics mode allows you to type reverse-video characters, in addition to the 20 special graphics characters. To see how this works, begin by typing the first part of the line:

```
70 PRINT AT 9,7;" **
```

(After the opening quotation mark, there is a space and then two asterisks. The asterisk is a shift character, located on the [B] key.) Next shift into the graphics mode (SHIFT-9), and type the word HELLO, without pressing the SHIFT key. You'll see that the entire word appears in white characters against a black background—that is, reverse video. Now type the space key; a reverse-video space is simply a black square. The space is the end of the message, so shift out of the graphics mode, and type the last two characters of the line—the ending quotation mark, and the semicolon. Press ENTER to make the line part of your program.

The next three lines of the program—80 to 100—make up another FOR/NEXT loop. By now, you should have no trouble typing the lines in. They use three new programming functions, so you'll have to shift into the function mode three times. Line 80 uses the LEN function, located beneath the [K] key:

```
80 FOR I = 1 TO LEN N$
```

The LEN function supplies the *length,* in characters, of a string. Line 90 uses the CHR$ function (beneath the [U] key) and the CODE function (beneath the [I] key):

```
90 PRINT CHR$(CODE N$(I) + 128);
```

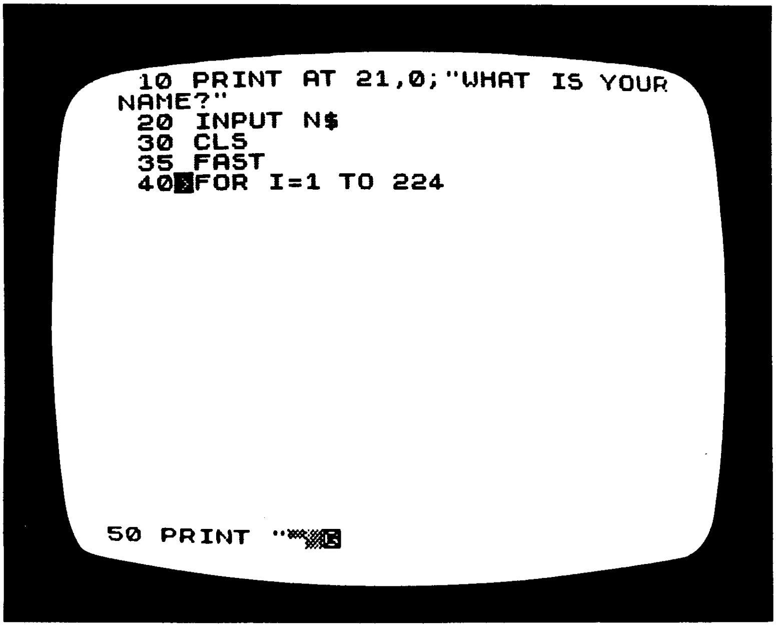
This is perhaps the most complex instruction in the program. Briefly, it converts the characters of the string N$ into their reverse-video equivalents. We will study the CHR$ and CODE functions in Chapter 5. Line 90 contains another new element: the plus sign (+). This is a shift character, located on the [K] key.

Line 100 ends the FOR/NEXT loop:

```
100 NEXT I
```

And, finally, line 110 is another PRINT instruction:

```
110 PRINT " **"
```



```
 10 PRINT AT 21,0;"WHAT IS YOUR
NAME?"
 20 INPUT N$
 30 CLS
 35 FAST
 40▮FOR I=1 TO 224




50 PRINT "▒▒▒▮
```

Figure 2.9: Typing the Graphics Characters

Notice that there are three characters within the quotation marks: first a space, then two asterisks.

You have now typed the entire program into the computer, and you're ready to see what it does when the computer performs its instructions. Before you run the program, though, do two things. First, compare your screen carefully, line by line, with the listing of the program shown in Figure 2.1. Make sure each character is exactly right. If you should find any errors, it's not hard to correct them. Skip ahead to the section of this chapter called "Editing Any Line of the Program," where you will see how to make additional changes in your program. (You already know how to use the editor; all you have to learn is how to specify which line you want to edit.)

The second thing to do before you run the program is review carefully all that you have learned about the modes of the keyboard. Study Figure 2.10, which summarizes all the different cues for you.

ACTION!

The keyword RUN, located above the [R] key, tells the computer to begin executing the instructions of the program in its memory. When you run this program that you've worked so hard to enter into the computer, you will probably recognize the source of at least part of the action in some of the more understandable lines you typed. At this point, since you have not studied BASIC yet, your idea of how the program works will be impressionistic rather than systematic. But still, you may be surprised at how accurately you can guess the meaning of most of the lines of the program once you have seen what they do.

Your Conversation with the Computer

Type the RUN keyword directly, without typing a number before it. Press the ENTER key and watch what happens.

For the first time, the listing of the program lines has disappeared from the screen. Of course, the lines are still stored in the computer's memory, but now the screen is needed to display the *results* of the program.

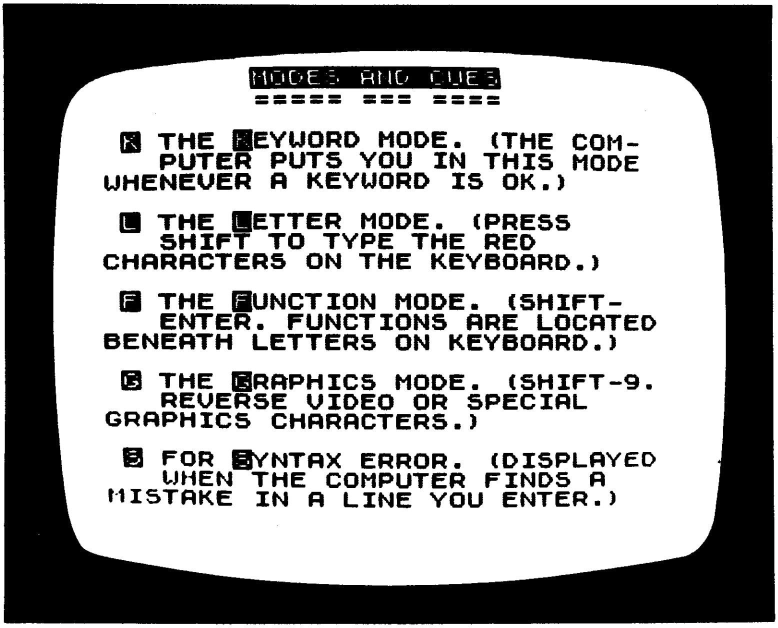
The action takes place in two distinct steps, as you will soon see. In the first step, the question:

```
WHAT IS YOUR NAME?
```

appears on the screen. The L cue, between quotation marks, is printed below the question. This tells you that the computer is waiting for you to type an answer on the keyboard. Go ahead and type your name. (Figure 2.11 shows how it will look, with the name HILDA.) If you make a mistake while you're typing, you can use the DELETE key (SHIFT-0) to erase the mistake, just as you did when you were typing the program.

When you have typed your name, press the ENTER key. The screen will go blank for a few moments, and then ... well, you'll see. (Figure 2.12 shows the greeting to Hilda.) Notice the clever way the computer has incorporated the initial of your name into the background design.

Before continuing, you should pause to reflect for a moment on the illusion created by your program. It looks as though *the computer* has asked you a perfectly civil question, has absorbed your answer, and now has produced a somewhat extravagant, but correct—and

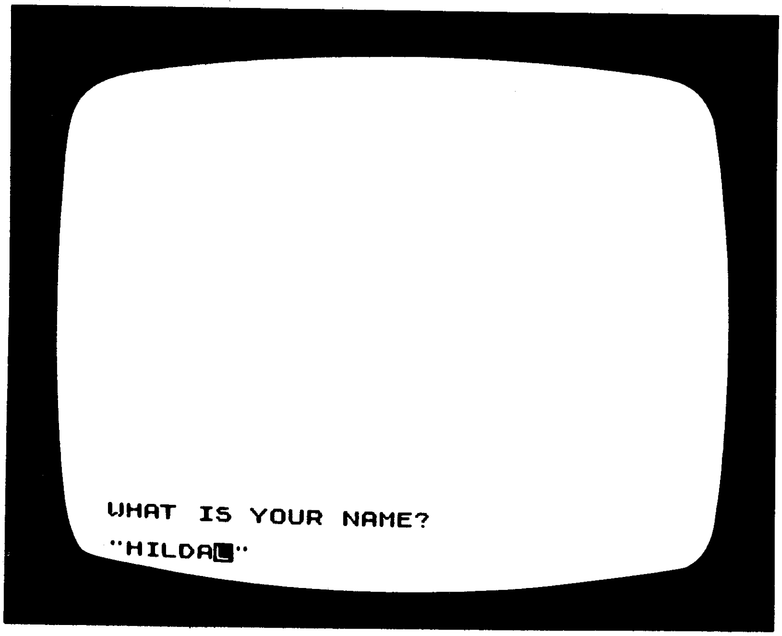


```
           MODES AND CUES
           ===== === ====

  K THE KEYWORD MODE. (THE COM-
     PUTER PUTS YOU IN THIS MODE
  WHENEVER A KEYWORD IS OK.)

  L THE LETTER MODE. (PRESS
     SHIFT TO TYPE THE RED
  CHARACTERS ON THE KEYBOARD.)

  F THE FUNCTION MODE. (SHIFT-
     ENTER. FUNCTIONS ARE LOCATED
  BENEATH LETTERS ON KEYBOARD.)

  G THE GRAPHICS MODE. (SHIFT-9.
     REVERSE VIDEO OR SPECIAL
  GRAPHICS CHARACTERS.)

  S FOR SYNTAX ERROR. (DISPLAYED
     WHEN THE COMPUTER FINDS A
  MISTAKE IN A LINE YOU ENTER.)
```

Figure 2.10: The Modes and Cues

seemingly intelligent—response. Of course, you know now that this "dialogue" was something of a fake, in that each detail of the action was completely controlled by the program you typed into the computer. The computer doesn't care a bit what your name is, and has no urge whatsoever to be friendly. But you can create the *illusion* of friendliness—or of any other kind of reaction—by the way you write your programs. What's more, the fact that a program exists at all, somewhere behind the scenes, is completely invisible to the person who answers the question displayed on the screen. (Just as an experiment, run the program again for someone else and see how the person reacts.) Creating this illusion of intelligence—of responding correctly to human input—is an essential part of what computer programming is all about. The computer can do nothing without your detailed instructions to guide it; but, ironically, if you learn to write clever programs, the cleverness will be attributed to the computer. No one will know that the program is there at all.

Returning to the TV screen, take note of a short message that appears in the lower left corner. The computer has finished



```
WHAT IS YOUR NAME?
"HILDA█"
```

Figure 2.11: Answer a Question on the Screen

executing all the lines of your program now, and the numbers in the corner represent a kind of status report on the results of the program:

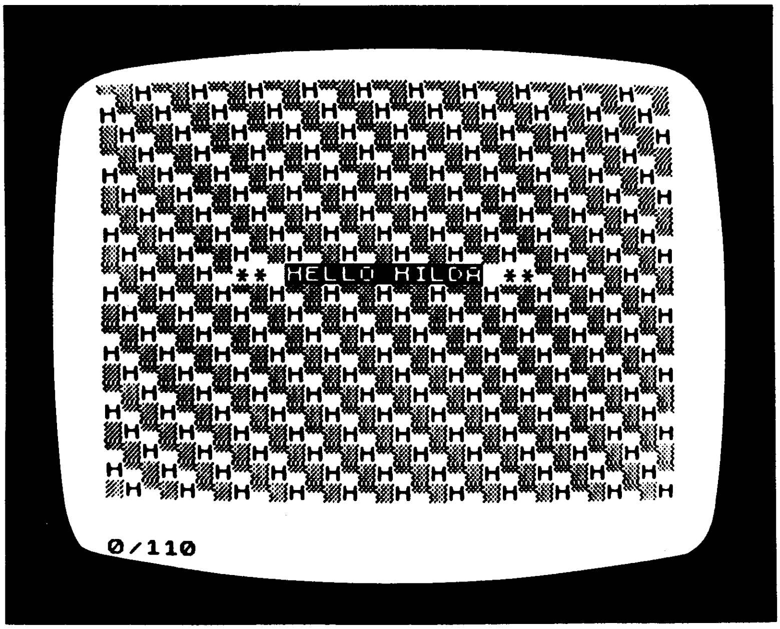
```
0/110
```

In later chapters we will look in detail at the kind of information this report can give you. In this case, the zero means that the entire program was completed, and no special problems arose during the performance. The number 110 tells you which line was the last to be executed.

EDITING ANY LINE OF THE PROGRAM

If you have finished admiring the computer's lovely greeting, press the ENTER key now. The greeting disappears, and the listing of the program returns to the screen. You can run the program as many times as you want—always by entering the keyword RUN; the computer will never get tired of it.

You can also revise the program, if you wish. You can do this



```
** HELLO HILDA **

0/110
```

Figure 2.12: The Computer's Greeting

in several different ways, depending on the kind of revision you want to make:

- You can add a new line, simply by typing it into the computer, making sure you give it a line number that isn't already taken by one of the other lines in the program.
- You can delete any line by typing its line number and then the ENTER key. (Knowing this, you should take care not to delete a line accidentally by typing a number that happens to be a line number in your program.)
- You can completely rewrite an entire line by typing a new instruction with the same line number. The old line will disappear, and the new one will take its place.
- Finally, if you have only a small change to make in a line, you can *edit* the line. We'll try doing this now.

You may have noticed a small detail in the program listing that we haven't mentioned up to now: the small reverse-video arrow that points to the last line you entered into the program. If you typed the lines of the program in sequence, this arrow now points to line 110. (You can see the arrow in Figure 2.1.) The arrow indicates the *current line*—that is, the line that will be edited if you press the EDIT key. You can move the arrow up and down the program listing by pressing the up and down direction arrows on your keyboard (SHIFT-6 and SHIFT-7).

Let's say, for example, that you would like to make a change in the greeting that the computer displays on the screen. You want to change the word HELLO to HOWDY. This part of the greeting is in line 70 of the program, so that's the line you want to edit.

If the current-line arrow is pointing to line 110, you need to move it up four lines in order to edit line 70. Here are the steps you will follow:

1. Press one finger down on the SHIFT key; then press the [7] key four times. Each time you press it, the listing will be reprinted on the screen and the arrow will move up by one line.
2. Now, with the arrow pointing at line 70, press the EDIT key (SHIFT-1) and a copy of the line will move down to the bottom of the screen, ready for editing.

3. Using the right-arrow key (SHIFT-8), move the L cue to the right of the letter O in HELLO. Press the DELETE key (SHIFT-0) four times to delete four letters.
4. Shift into the graphics mode (SHIFT-9) and type the new letters —OWDY.
5. Finally, shift back out of the graphics mode (SHIFT-9 again) and press the ENTER key. The revised version of line 70 will appear as part of the program listing.

Run the program now to make sure it still works. Everything should be the same, except that HOWDY will replace HELLO in the greeting.

An additional note about moving the current-line arrow in the program listing: When you are working with a very long program, you may want a faster way to move the arrow from one line to another, especially if the lines are far apart. In this case you can use the keyword LIST. Let's say you want to move the arrow to line 40. Enter the command:

```
LIST 40
```

Your program will be listed on the screen from line 40 on. Now type the EDIT key, and a copy of line 40 will be moved to the bottom of the screen for editing.

FOR THE NEXT PERFORMANCE . . .

In Chapter 1 we discussed the need for a cassette tape recorder to provide a permanent storage medium for programs. You'll recall that programs are stored in the computer's temporary memory, and thus are lost when you unplug the computer.

In this section, you'll learn how to use a cassette recorder with your computer. If you have a recorder, get it ready. We'll go through a practice recording session with the greeting program.

Saving Your Program on Tape

Find the short cord that has a pair of plugs on either end. The reason it has two plugs is to allow you to connect both the earphone and microphone jacks of your tape recorder to your computer at once. In fact, however, it's probably safer to connect only one set of jacks at a time. Plugging them both in together can sometimes interfere with the recording process. The plugs are two different colors so that you can easily match up the plug on one end with the correct plug on the other end.

To record a program onto a cassette from your computer, first connect the jack labeled "MIC" (on the left side of the computer) to the microphone jack on your tape recorder. Make sure you plug both ends of the cord tightly into the correct jacks. (If you find that the plug fits loosely into your computer, you may have to hold it manually in the jack during both the recording and play-back sessions.)

The volume and tone settings on your tape recorder are important. Start with the volume set at about half the maximum. Put the tone setting at the highest treble and/or the lowest bass. If your first attempt to record a program is not successful, try turning the volume up.

Put a fresh cassette in your tape recorder, and rewind it to the beginning if necessary. If your tape recorder has a counter, set it at zero so you can keep track of the beginning and ending points of the recording.

The keyword that you use for saving a program onto a cassette is SAVE, located above the [S] key. The SAVE command requires a program title, typed between quotes. You should think of a title that will remind you of exactly what the program does. We'll call this program GREETING. Type the command, but don't enter it yet:

```
SAVE "GREETING"
```

Now press the RECORD button on your tape recorder, let it run for a few seconds, and then press the ENTER key on your computer. Your screen will go blank for a few moments. When you see the screen fill with black horizontal stripes (about one inch thick) that shake quickly up and down, you'll know that the program is being sent out to the tape recorder. The length of time the stripes are on the screen depends on the length of the program. The greeting program will only take a few seconds to record. Very long programs can take many minutes.

When the recording is complete, the screen will go blank again except for the message:

```
0/0
```

displayed in the lower left corner.

Now you can actually use your tape recorder to listen to the recording if you want to, to make sure that everything went all right. The recording of the program will produce a high, unpleasant screeching noise, reminiscent of a burglar alarm. You don't have to listen to it for very long. If you hear it starting, turn off your tape recorder; the recording was probably successful. (If the noise isn't there, then you'll have to try the recording process over again.)

You should always make two copies of any important programs on two different cassettes. Put one of them away in a safe place; it will be your "backup" copy in case something should happen to the copy you use regularly. On each cassette, write the name of the program, and, if possible, its numerical location on the tape.

To play a program back from a cassette to your computer, you must connect the jack labeled "EAR" on your computer to the output jack on your tape recorder. (The jack on your recorder might be labeled "monitor," "extension speaker" or "earphone.") Again, it's important that the plugs be securely connected.

The LOAD command, located above the [J] key, is used for receiving a program into the computer from a cassette. For the greeting program, type the command:

```
LOAD "GREETING"
```

If you don't know the name of a program, you can also type the command without the title, but the two quotation marks are still required:

```
LOAD ""
```

Make sure the cassette is rewound to the correct position, and press the PLAY button on your recorder. Then press the ENTER key on your computer. At first you'll see a diagonal pattern of thin (about one eighth to one quarter inch), relatively stable black lines on your screen. Then, when the program starts to play, you should see the same heavy black stripes that appeared when you were recording the program. When the cassette reaches the end of the program, your TV screen should go blank. Again, you'll see the message:

```
0/0
```

in the corner of the screen. This indicates a successful playback of the program. Press the ENTER key and the program will be listed on the screen.

If the stripe pattern doesn't stop when it should, something has gone wrong. Turn off the tape recorder and press the SPACE key on your computer. (Notice that the word BREAK is written above this key. You can always press BREAK to interrupt a program run, or a saving or loading process.)

The most likely causes of failure when you're trying to save or load a program are:

1. Faulty connections. (Check to see if the plugs are pushed snugly into the correct jacks.)
2. Incorrect volume or tone settings. (You may have to experiment with the best volume setting for your tape recorder.)

PURCHASING SOFTWARE

Mastering the tape recorder connection to the point where you can record and reload programs easily and reliably may take a certain amount of patience and perseverance on your part. But the bonus reward for your efforts, in addition to permanent storage of your own programs, will be the ability to use programs written and recorded by other people. In particular, you will be able to buy programs—some of them rather sophisticated—that may give you new perspectives on the use of your computer.

The amount of software available on cassette for the T/S 1000 is growing rapidly. (*Software,* by the way, is the word we use to distinguish computer programs from the *hardware* components of the computer system.) In such general-interest categories as business, home finance, education, and games, you will find a good selection of programs to choose from. Many of these programs require the 16K extra memory module, but some can be used on the basic computer unit without extra memory.

To give you an idea of the kind of program you will be able to buy, we'll look at an example in the next section of this chapter. We'll review a program called VU-CALC, which is likely to be one of the most popular of the programs available for the T/S 1000.

VU-CALC, A Spreadsheet Program

You may have heard or read about the family of software called *spreadsheet programs*. These programs allow you to organize large quantities of numerical data, and perform many calculations on them quickly and efficiently. Spreadsheet programs are now available for most of the larger personal computers. Two examples of such programs are VisiCalc® and SuperCalc™, both of which are phenomenally popular among small computer users. Spreadsheet programs will simplify and speed up any computing task that involves keeping track of, and performing arithmetic operations on, large sets of numeric data.

VU-CALC, designed for the T/S 1000 with the extra memory module, is the "kid brother" of these programs. Although it is neither as powerful nor as versatile as the spreadsheet programs designed for larger computers, it is a worthwhile programming tool for anyone faced with the drudgery of large arithmetic tasks.

Figure 2.13 shows a simple example of a VU-CALC spreadsheet. VU-CALC gives you a vast empty worksheet, 36 columns by 26 rows, into which you can type words, numbers, or formulas. Each position on the worksheet is referred to by a two-coordinate "address": a letter (indicating the row) and a number (indicating the column). So, for example, position F01 on the spreadsheet in Figure 2.13 contains the word NET. Positions F02 and F03 contain net income amounts for the years 1981 and 1982, respectively.

Once you have begun typing numbers on your spreadsheet, you can create additional data by writing formulas that refer to data already on the sheet. For example, positions G02 and G03 show income tax amounts calculated from the net income values contained in positions F02 and F03, respectively. In the lower-left corner of the screen you can see the formula that was used to calculate the value in position G03:

```
F03 * .35
```

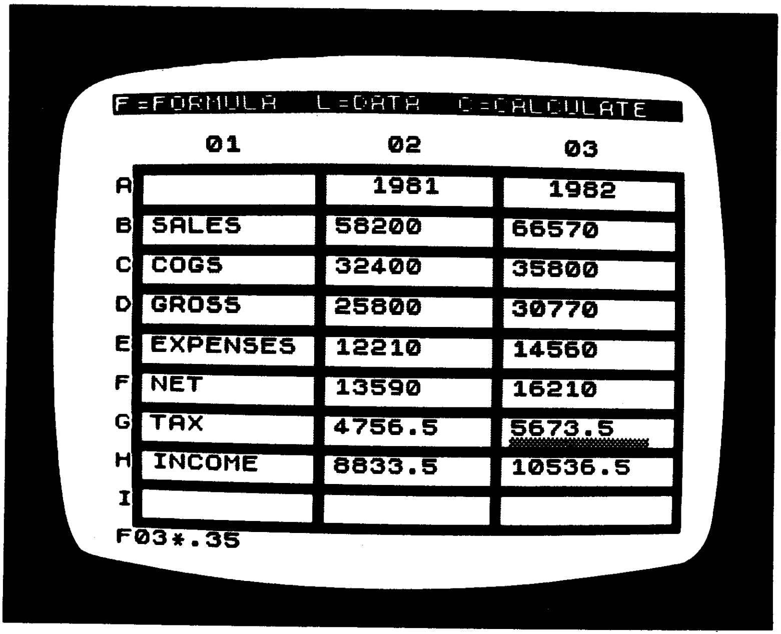
(The asterisk in this formula represents multiplication. The formula is shown on the screen because the worksheet's *cursor* is currently located at G03.) You only need to write such a formula once, and then you can apply it, if you wish, over many rows or columns of data. Most important of all, such formulas are dynamic; they are recalculated whenever any data on the worksheet changes.

While you only see three columns and nine rows of the worksheet at a time, the TV screen acts as a moveable "window" over the entire worksheet. To view other parts of the worksheet, you can "move the window" by pressing the four direction keys located at the top of your keyboard.

VU-CALC also lets you save spreadsheet data on cassette, along with the program. This way you can keep permanent records of the data you put on important spreadsheets.

A Consumer's Guide to Buying Software for the T/S 1000

Unfortunately, the cassette software available for your computer is relatively expensive. If you buy half a dozen programs, you'll spend about as much on the software as you originally paid for the computer. For this reason, you should learn to be a careful, selective, and well-informed consumer when you set out to purchase software. You should find a dealer who will let you try out a program before you buy it. (Many dealers have "floor samples" of all the software they sell, available for your inspection.)



```
F=FORMULA   L=DATA   C=CALCULATE
        01           02           03
A                    1981         1982
B  SALES             58200        66570
C  COGS              32400        35800
D  GROSS             25800        30770
E  EXPENSES          12210        14560
F  NET               13590        16210
G  TAX               4756.5       5673.5
H  INCOME            8833.5       10536.5
I
F03*.35
```

Figure 2.13: Example of VU-CALC

Each program comes with "documentation"—a small pamphlet describing how to use the program. Take the time to read the documentation of a program you are considering buying. The documentation should describe clearly and simply everything you need to know to run and use the program successfully. If the documentation is bad, you may spend so much time trying to understand the program that the benefit of your purchase will be lost.

Here are some questions you should ask yourself before you buy a new piece of software:

- If the program is oriented toward business or home finance, will it be useful to you in many different situations? Does it perform specific computing tasks that you need to do? (As you become more proficient in BASIC, you will be able to write many programs yourself, designed precisely for the jobs you need to accomplish.)
- If the program is a game, is it one that you'll enjoy playing time after time?
- If you have questions about how to use the program, will you be able to find the answers—either from the documentation, or from your dealer? When you try out the program in the store, does it seem to do what the documentation says it will do?
- How does the program react when you make a mistake? Does it provide you with easy ways to correct mistakes? You should make intentional errors while you are trying out the program and note the results.

Questions like these will help you decide whether or not a particular program is worth your time and money.

SUMMARY

Learning to use the keyboard of your computer is easy once you've mastered the various modes the keyboard shifts into. Since each line of a BASIC program must begin with a keyword, the computer always starts you out in the K mode. Following the keyword, an instruction line may include letters, numbers, or shift characters (the L mode); graphics or reverse-video characters (the G mode); or functions (the F mode).

Your computer has several important ways of helping you to recognize and correct any errors that you might make while typing a program. If you try to enter a line that contains a syntax error, the computer will reject the line, displaying it at the bottom of the screen with the S cue marking the error. To correct the line you can use the left- and right-arrow keys to move the cue to the proper position in the line, and you can insert or delete characters. On the other hand, if you enter a syntactically correct line that you subsequently wish to revise, you can use the computer's EDIT feature to make the change.

Mastering the cassette tape recorder connection with your computer may require a certain amount of trial and error on your part. However, once you have learned to use a cassette recorder for saving and loading programs, your computer will become a much more valuable tool. You will be able to begin weighing the relative advantages of purchased software, and you will begin building a library of programs that you have written yourself.

Chapter 3

The Plot Thickens: A Short, Graphic Course in BASIC

INTRODUCTION

In this chapter you'll learn the essentials of BASIC, while having a bit of fun with the graphics features of your computer. The programming skills you'll acquire in this chapter do not, of course, apply exclusively to writing graphics programs. You'll use the same commands and techniques later, when you begin writing other kinds of programs. But the graphics exercises in this chapter present an enjoyable way to begin learning BASIC. At every point you'll be able to *picture* exactly what the computer is doing in response to your instructions.

The T/S 1000 has two distinct systems for producing graphics on the screen. You saw an example of one of them in Chapter 2, with the use of the special graphics characters and the display of reverse-video letters. We'll begin this chapter by investigating both systems in detail. Then we'll move briskly through a short course in BASIC, covering the commands that you'll use most often in programs. You'll see illustrations of each command in a number of small programs, and, to summarize what you've learned, you'll work through a short graphics program that draws rectangles on the screen.

The final program in this chapter is a relatively long one, though it does fit into the 2K of memory provided by your computer. You'll be able to use it to draw many kinds of pictures and designs on the screen. The pictures will get better and better as your skill in using this program increases. The program itself merits careful study, as it illustrates several important techniques that will help you write clearer, more efficient programs.

IMMEDIATE COMMANDS

In Chapter 2 we discussed three characteristics of a BASIC program:

- the lines are numbered;
- each line begins with a keyword;
- the computer merely *stores* the lines, without performing any of the instructions, until you type RUN.

It happens that your computer also allows you to use many (but not all) of the keywords as *immediate* commands. That is, you can type an instruction, without a line number before it, and the computer will perform the instruction immediately, without waiting for you to type RUN. Sometimes you'll want to use immediate commands to explore the computer's reactions to one instruction or another.

For example, enter the following command into your keyboard:

```
PRINT "T/S 1000"
```

The result, you will see, is that the computer immediately prints T/S 1000 at the top of the screen. Now if you press ENTER again, you'll see that the computer has *not* saved this PRINT instruction. Immediate commands are performed only once, and then are lost.

You've actually already seen several other examples of immediate commands. You know that SAVE and LOAD are used for storage and retrieval of programs on cassettes. LIST tells the computer to display the lines of the program on the screen. RUN makes the computer begin performing the lines of the program. Another command you may have occasion to use soon is NEW, which clears the current program out of the computer's memory so you can write a new program. All these instructions are usually given as immediate commands—although they can also be used as lines in programs.

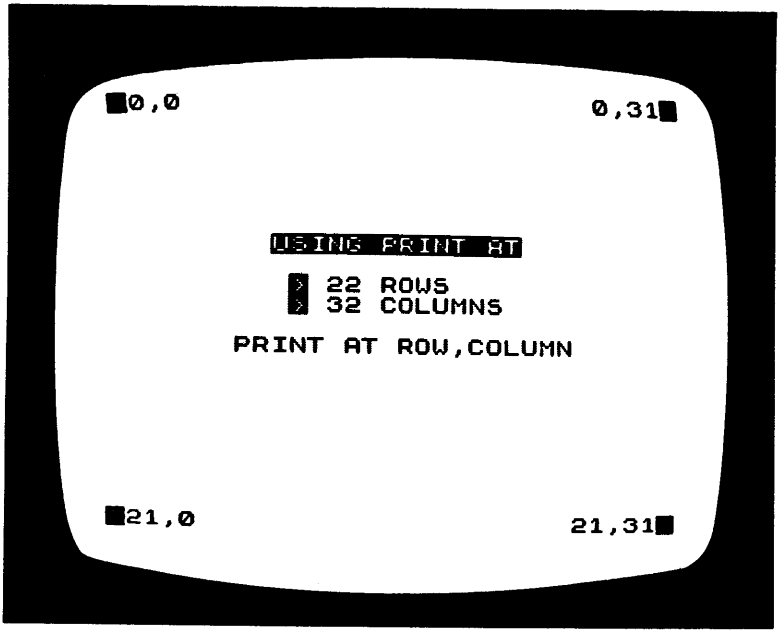
Other commands, such as PRINT, are normally used as part of a program, but they can be executed as immediate commands. We will use PRINT as an immediate command in the next section of this chapter in order to carry out several quick experiments.

TWO SYSTEMS OF GRAPHICS

The PRINT AT Instruction

When you use the PRINT command to display information on the screen, all the characters you print take up an equal amount of space. Think of the screen area as a grid, divided into rows and columns; you can put one character into each square of the grid. The screen area is 22 rows long and 32 columns wide. (The bottom part of the screen, below the 22nd row, is reserved by the computer as work space. As you know well by now, this is the area where new lines appear as you type them in. You can't access this area at all with the PRINT command.)

Using the PRINT AT instruction, you can specify exactly where you want information to appear on the screen. As you saw in Chapter 2, PRINT AT is always followed by two numbers. These numbers represent an "address" on the screen. Figure 3.1 will help you understand how these addresses are determined. The first



```
█0,0                           0,31█



         USING PRINT AT
          █ 22 ROWS
          █ 32 COLUMNS
      PRINT AT ROW,COLUMN




█21,0                         21,31█
```

Figure 3.1: Using the PRINT AT Instruction

number following PRINT AT is the *row,* and the second number is the *column* of the address. Thus, the correct form of the PRINT AT instruction is:

```
PRINT AT row, column; "some information"
```

Notice once again that the numbers representing the row and column are separated by a comma, and that the address is separated from the information by a semicolon.

Now look again at Figure 3.1. The addresses begin at the upper-left corner of the screen. You can think of this position as the *origin,* if you like. Its address is 0,0. Notice the addresses of the other three corners of the screen. The row numbers increase as you go down the screen, and the column numbers increase as you go across to the right.

To make sure you understand this address system, study the following instructions, and try to determine approximately where each of them will put a message on the screen:

```
PRINT AT 19,5; "X"
PRINT AT 7,7; "WHERE WILL THIS GO?"
PRINT AT 15,25; "HELLO"
```

Now enter them as immediate commands, one at a time, and see if you predicted correctly what their results would be. Notice that when the message contains more than one character, it is the *first* character that will be located at the specified address; the rest of the characters follow on the same line.

You can use a simple PRINT instruction to display several messages at several different locations. Try the following command:

```
PRINT AT 3,3; "HERE"; AT 20,25; "THERE"
```

In summary, PRINT AT can be used to print any character, word, number, or special graphics character at any position on the 22-by-32 screen.

PLOT and UNPLOT

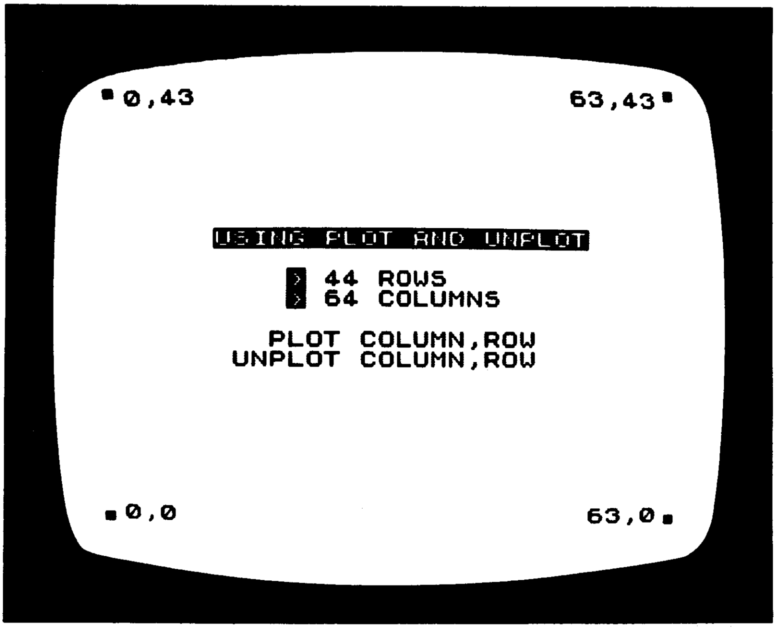
Your computer also gives you a second method of sending graphics to the TV screen, using the keywords PLOT and UNPLOT. These commands organize the screen somewhat differently, and require a different system of addresses.

Figure 3.2 summarizes the use of PLOT and UNPLOT. The PLOT command produces a small black square, called a pixel (for "picture element") at a specified address on the screen. Each performance of PLOT puts a single pixel on the screen. The four corners of the screen in Figure 3.2 contain pixels. The screen area can display 44 rows by 64 columns of pixels. Recall the grid address system of the PRINT AT instruction. If you were to divide each square of this grid into four smaller squares, you would see the size of a pixel.

The origin of the pixel addresses is in the lower-left corner of the screen, rather than the upper-left. This position has the address 0,0. The row addresses increase to 43 as you move up the screen, and the column addresses increase to 63 as you move across the screen to the right. The general form of the PLOT instruction is:

```
PLOT column, row
```

To become familiar with pixels and the system of addresses used for the PLOT command, enter the following lines, one at a time, as immediate commands, and notice where each one places



```
▪0,43                                63,43▪



            USING PLOT AND UNPLOT
                 > 44 ROWS
                 > 64 COLUMNS

               PLOT COLUMN,ROW
             UNPLOT COLUMN,ROW



▪0,0                                  63,0▪
```

Figure 3.2: Using the PLOT and UNPLOT Commands

its pixel:

```
PLOT 5,5
PLOT 55,3
PLOT 40,40
```

The UNPLOT instruction "erases" a pixel from the screen. It takes the same form as the PLOT command:

```
UNPLOT column, row
```

We'll be seeing a lot of the PRINT AT, PLOT and UNPLOT commands in the programs of this chapter, so make sure you understand how they work before you read on. If you are still uncertain about the two different systems of organizing the screen, review Figures 3.1 and 3.2 and keep experimenting with immediate commands.

A QUICK COURSE IN BASIC

This section will introduce you to most of the essentials of BASIC programming for the T/S 1000: defining variables; reading input information; repeating instructions; controlling the order of performance; and making decisions. You'll find many short sample programs that will illustrate each point. It's important that you enter each of these programs into your computer and try them out as you read. If you don't understand why they produce the results they do, go back and review the explanation given.

Variables Revisited

We've seen that a variable is simply a place in the computer's memory reserved for a certain type of information. You can define many different variables for use in a program. So that you can access the information easily, each variable has a name. The information contained in a variable may change, but the name remains the same. We have also seen that there are two *types* of variables—numerical and textual, or *string*.

You can define variables—or change the value contained in a variable—with the LET statement. The general form of the LET statement is:

```
LET NAME = VALUE
```

This statement, paraphrased, means "store VALUE in a variable called NAME." When the computer performs a LET statement, it first evaluates the information on the right side of the equal sign, and then stores that information in the variable named on the left side of the equal sign.

Let's look at some examples. The LET statement can be used as an immediate command, so you can enter the following lines directly, without line numbers. Let's say you want to define a variable called INCOME to store the amount of income you earned this year. Your command might be something like this:

```
LET INCOME = 15000
```

Enter this command and note the results. You see only the message 0/0 in the lower left corner of the screen—meaning that the computer has performed the command.

Now to assure yourself that INCOME has been defined, enter:

```
PRINT INCOME
```

You will see that the *value* of the variable INCOME—15000—is printed at the top of the screen. So, when you include a variable name in a PRINT statement, the computer displays the *value* contained in that variable on the screen.

You are free to choose any name you want for a variable. Whenever possible, it's best to choose a name that will describe what the variable stands for. In the case of *numerical* variables, it's also up to you to decide on the length of the variable name. Many people prefer to use long variable names, such as INCOME, because they make a program easier to read. You can see right away what the variable is for. The disadvantages of long variable names are, first, that they take longer to type at the keyboard, and second, that they take up more of the computer's memory. For these reasons, you may find yourself writing shorter, abbreviated variable names, such as INC, or even simply I.

Once you've defined a variable with a certain name, the computer will only recognize the exact name you've specified. If you accidentally misspell a variable name, or if you try to abbreviate it once it's already been defined, the computer won't know what you mean. For example, now that you have defined the variable INCOME, try entering each of the following PRINT lines:

```
PRINT IMCOME
PRINT INC
PRINT I
```

None of these names can be used to access the information in the variable INCOME. After each one of these commands, you'll see the message:

```
2/0
```

in the lower-left corner of the screen. The 2 is a code for one of the computer's error messages; it means simply that you have used an undefined variable name. (You can see the codes for all the error messages in Appendix B.)

In a program, variables are your means of storing pieces of information that you will need to use again. In addition to numerical information, you may also want to store characters, words, or text in the computer's memory. As we've seen, items of *nonnumerical* information are called strings, and the variables that hold them are called string variables. The name of a string variable, you'll recall, must end in the character "$". Unlike numerical variable names, the names of string variables must be exactly two characters long: a letter and the $ character.

Here are some examples of LET statements that define string variables:

```
LET A$ = "2344 SIXTH STREET "
LET C$ = "BERKELEY "
LET S$ = "CA"
```

Enter these three LET satements and then type the command:

```
PRINT A$; C$; S$
```

The result will be:

```
2344 SIXTH STREET BERKELEY CA
```

printed at the top of your screen. This shows you something new about the PRINT statement; it can contain as many variable names as you wish. The computer will simply display the contents of each variable on the screen. Notice that each variable is separated by a semicolon in the PRINT statement. This tells the computer to display the contents of the variables side by side on the same line.

Now try these two PRINT statements:

```
PRINT "I LIVE IN "; C$;"."
PRINT "MY INCOME IS $"; INCOME;"."
```

These lines show you that a PRINT statement can include any combination of elements; literal string messages, numerical variable names, string variable names. The first of these two statements should put the message:

```
I LIVE IN BERKELEY.
```

at the top of your screen. What does the second statement do? As a further experiment, use LET commands to change the value of C$ to the name of your city and the value of INCOME to your income. Then enter these two PRINT statements a second time and watch the results.

There are several ways to clear variables—and the data they contain—out of the memory of your computer. The easiest way, of course, is to unplug the power, but if you do that you will lose everything, including the program you might be working on. Instead, you can use the keyword:

```
CLEAR
```

which is located above the [X] key. This command clears all variables and data from memory, but leaves all program lines intact. Type this command now, and then enter:

```
PRINT INCOME
```

You'll get the error message 2/0, telling you that the variable INCOME is no longer defined.

Finally, each time you run a program, using the keyword RUN, the variables and data left over from any previous runs of the program are cleared from memory.

A second way to define variables, in addition to the LET statement, is with the INPUT statement. We'll study this statement next.

READING INFORMATION FROM THE KEYBOARD

In T/S 1000 BASIC, the INPUT statement has only one form:

```
INPUT name
```

where "name" can be any variable name.

As you saw in the greeting program of Chapter 2, INPUT tells the computer to wait for some data to be typed onto the keyboard, and then to store that data in the variable named in the INPUT statement. This variable can either be one that hasn't yet been defined, or it can be one that already has a value. In the latter case, the old value is lost and the new value, read from the keyboard, takes it place.

INPUT cannot be used as an immediate command, so we'll have to write a short program to experiment with it. Type in the following six lines:

```
10 PRINT AT 21,0; "NAME?"
20 INPUT N$
30 PRINT AT 21,0; "AGE? "
40 INPUT A
50 CLS
60 PRINT N$; ", AGE ";A
```

When you run this program, you will see that the action stops twice for you to enter information at the keyboard. First type your name, and then your age. When you are finished, a message like this will appear at the top of the screen:

```
JOHN DOE, AGE 31
```

There are several important things to learn from this program and the action it results in. First of all, what exactly does the computer display on the screen when it is waiting for input from the keyboard? If you were watching closely during the program run, you noticed two different kinds of displays on the screen. When the computer is waiting for a string, it displays the L cue (in reverse video, of course) between quotation marks. When it is waiting for numeric data, on the other hand, it simply displays the L cue alone. This is how the computer distinguishes between an INPUT statement that contains a string variable name and one that contains a numeric variable name.

Lines 10 and 30 in this short program illustrate an important design feature for clear and efficient programming. Whenever you write a program that has INPUT statements, you should think carefully about the person who will be using the program. That person will need to be told what information to type into the computer when the action stops for input. For this reason, it is a good idea to print a "prompt" on the screen whenever the computer is waiting for input. Lines 10 and 30 give examples of such prompts. Notice that these lines place their prompts down near the bottom of the screen. This is done so that the prompt will appear as close as possible to the "echo" of the information that is being entered at the keyboard.

Line 50 of this program clears the screen, so that the message of line 60 will appear at the top of the screen. Study line 60 carefully; notice that the message between quotes (", AGE ") must supply a comma and two spaces to separate the information from the two variables:

```
60 PRINT N$; ", AGE ";A
```

To summarize, we have seen that variables can be defined in either of two ways: with the LET statement, or with the INPUT statement. In general, you cannot use a variable in a program unless it has been defined in one of these two ways. (There is one exception to this rule, which we will see in the next section of this chapter.) Once a variable has been defined and given a value, you can use a PRINT statement to display its value on the screen. You can also *change* the value contained in a variable with subsequent LET and INPUT statements. We will continue to investigate this subject as this chapter progresses.

Before we move on, let's look at one more short program illustrating the use of variables. Type the keyword NEW to clear the last program from your computer's memory. Now type in the following five lines:

```
200 PRINT AT 21,0; "HORIZONTAL"
210 INPUT H
220 PRINT AT 21,0; "VERTICAL"
230 INPUT V
240 PLOT H,V
```

These lines are actually part of a larger program that we'll be building as we go along, but you can run these five lines as though they formed a complete program. When you run the program you'll see two input prompts on the screen, first:

```
HORIZONTAL
```

at which point you should enter a number between 0 and 63; then:

```
VERTICAL
```

to which you should respond with a number between 0 and 43. When you have entered both numbers, the program will display one pixel on the screen at the address that you specified with the two numbers.

Make sure you understand how the program works. Lines 210 and 230 contain the INPUT instructions, defining the numeric variables H and V, respectively. Then line 240 uses the values in H and V as the address in a PLOT command:

```
240 PLOT H,V
```

The PLOT and UNPLOT instructions do not require literal numbers for the pixel address. They can instead take variables for the address, as long as the variables are defined, and as long as the values they contain form a legal pixel address.

Next we will investigate the FOR/NEXT statements; you'll recall from Chapter 2 that these statements can be used to make the computer repeat instructions many times.

REPEATING INSTRUCTIONS

An example of the simplest form of the FOR statement is:

```
FOR I = 1 TO 20
```

The I in this statement can be called the *control variable,* because it controls the number of times that the FOR statement, and the statements after it, will be repeated. The name of the control variable must be only one letter long. The control variable does *not* need to be defined in advance by a LET or INPUT statement. (This is the exception to the rule.)

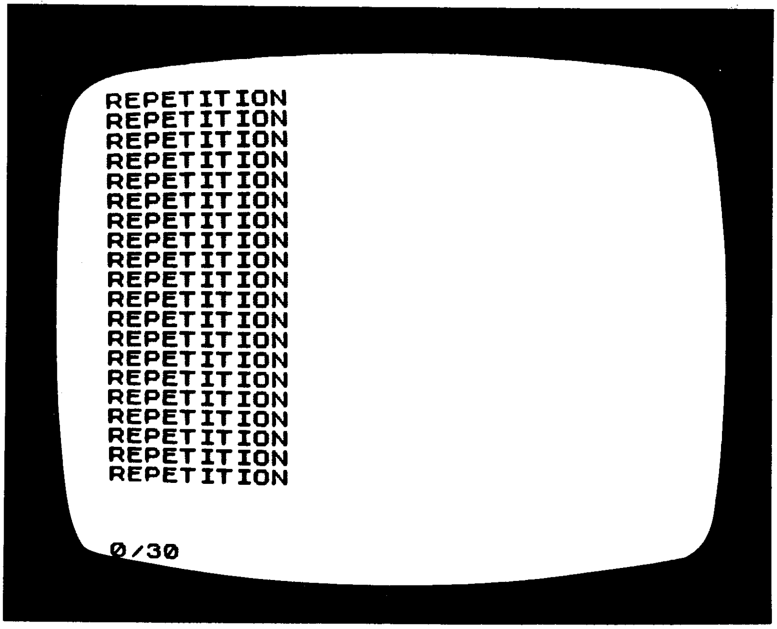
Let's look at the FOR statement in action. FOR and NEXT cannot be used as immediate commands, so we will look at a short program:

```
10 FOR I = 1 TO 20
20 PRINT "REPETITION"
30 NEXT I
```

Remember that the FOR and NEXT lines represent the top and the bottom of a "loop." All the lines in between will be repeated a specified number of times. Here is how the computer uses the control variable, I in this case, to decide on the number of repetitions:

1. At the beginning, I is given the value that appears just after the equal sign, 1 in this case.
2. All the statements between FOR and NEXT are performed.
3. When the computer gets to the NEXT line, it increases (or *increments*) the value of I by 1, and then refers back to the FOR line.
4. If I is still *less than or equal to* the number specified *after* the word TO (20 in this case), the computer repeats all the lines up to NEXT again. If I is greater than the number that appears after TO, then the looping stops.

Type these three lines into your computer, and run them. Figure 3.3 shows the results. As you can see, the PRINT instruction in line 20 has been performed 20 times. In other words, the control



```
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION
REPETITION

0/30
```

Figure 3.3: Repetition

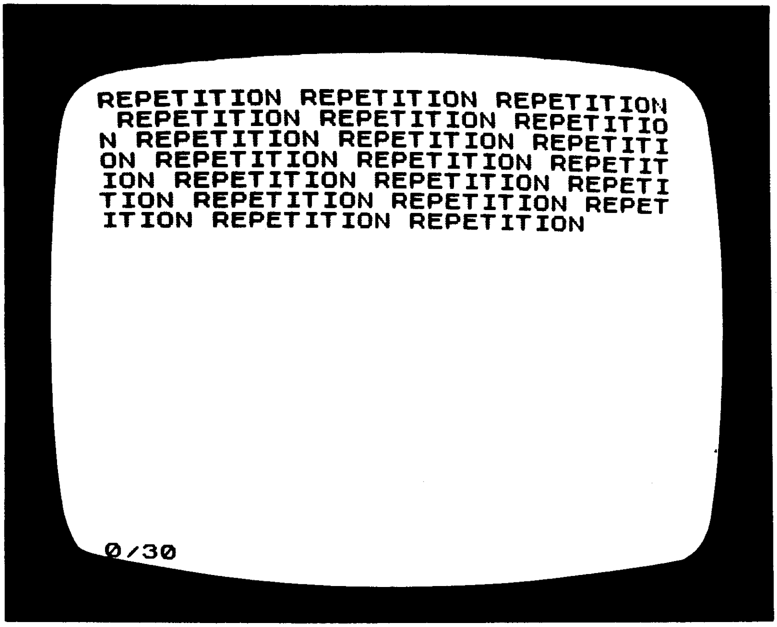
variable I has been incremented from 1 to 20; after each incrementation the PRINT instruction was performed one time.

To further explore the repetition concept, we will look at two variations of this short program, each of which involves making small changes in line 20. To make these changes, use the EDIT feature of your computer.

First, change line 20 so that it looks like this:

```
20 PRINT "REPETITION ";
```

Two changes have been made: a space has been added between the N and the closing quotation mark, and a semicolon has been added to the end of the line. When you've made these changes, run the program again; the results should look like Figure 3.4. This little exercise is simply to remind you of the function of the semicolon in a PRINT statement. The semicolon tells the computer *not* to go to a new line. (We say that the semicolon prevents a *line feed.*) Thus, when a PRINT statement *ends* in a semicolon, the next PRINT



```
REPETITION REPETITION REPETITION
 REPETITION REPETITION REPETITIO
N REPETITION REPETITION REPETITI
ON REPETITION REPETITION REPETIT
ION REPETITION REPETITION REPETI
TION REPETITION REPETITION REPET
ITION REPETITION REPETITION




0/30
```

Figure 3.4: More Repetition

statement that is performed will start off where the last message ended.

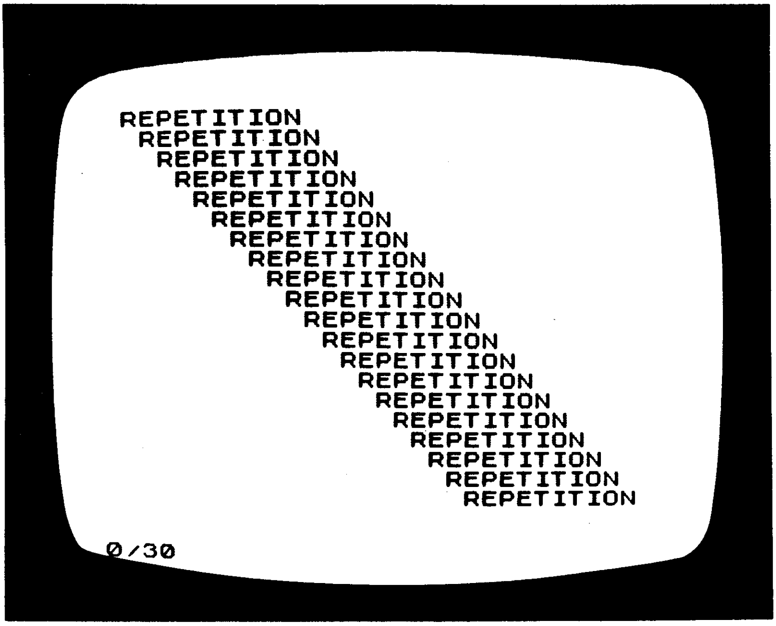
As a final experiment, change line 20 as follows:

```
20 PRINT AT I,I; "REPETITION"
```

and run the program a third time. The results are shown in Figure 3.5.

This last version of the program illustrates an essential point: *The control variable of the FOR statement can be used as a variable in other kinds of statements that appear inside the loop.* The fact that the control variable receives an incremented value each time the loop is repeated can make this variable an extremely valuable tool for certain kinds of repetitive actions. In Figure 3.5 you can see that the control variable, I, determined the address for each performance of the PRINT AT statement. As the value of I was incremented from 1 to 20, the PRINT AT statement was executed as:

```
PRINT AT 1,1; "REPETITION"
PRINT AT 2,2; "REPETITION"
```



```
REPETITION
 REPETITION
  REPETITION
   REPETITION
    REPETITION
     REPETITION
      REPETITION
       REPETITION
        REPETITION
         REPETITION
          REPETITION
           REPETITION
            REPETITION
             REPETITION
              REPETITION
               REPETITION
                REPETITION
                 REPETITION
                  REPETITION
                   REPETITION

0/30
```

Figure 3.5: Still More Repetition

```
PRINT AT 3,3; "REPETITION"
PRINT AT 4,4; "REPETITION"
```

. . . and so on.

Let's continue to investigate this last point a little further with another example. Revise your three-line program so that it looks like this:

```
10 FOR I = 20 TO 40
20 PLOT I,I
30 NEXT I
```

In this version, we'll use the value of the control variable to determine the addresses of pixels produced by the PLOT command. Run the program, and you'll see a diagonal line of pixels running from the address 20,20 to the address 40,40. Notice that the control variable, I, does not start at 1 in this program; instead it is incremented from 20 to 40 by the specification in the FOR statement.

By changing line 20 to:

```
20 PLOT I,20
```

you can produce a horizontal line of pixels. Likewise, the line:

```
20 PLOT 20,I
```

will produce a vertical line of pixels when you run the program. You should try each of these versions of the program.

In all the examples we have seen so far, the control variable has been incremented by 1 for each repetition of the FOR loop. In some cases you will want to change the *amount of incrementation* to some number other than 1. You can do so with the word STEP; for example:

```
FOR I = 2 TO 40 STEP 2
```

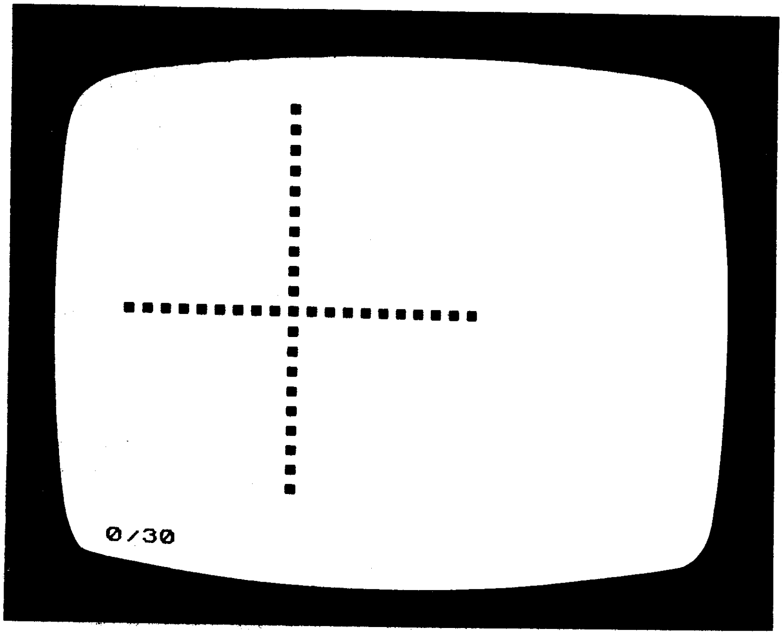
(STEP is a shift character, located on the [E] key.)

To see how STEP works, study the following version of our program:

```
10 FOR I = 2 TO 40 STEP 2
20 PLOT I,20
25 PLOT 20,I
30 NEXT I
```

In this program, the control variable, I, will take the values 2, 4, 6, 8, . . ., 40. As a result, the PLOT statements will produce broken lines, because every other address along the line will be skipped. (This program also shows how easy it is to add lines to a FOR loop. Line 25, the second PLOT statement, produces a second line on the screen.) Run the program. Figure 3.6 shows what will appear on the screen. Study this program and its results carefully before you move on. Make sure you understand why it does what it does.

In all the programs we've looked at so far, the order in which the lines are performed is determined solely by the line numbers. The computer starts with the smallest line number, and moves up, line by line, until the last line of the program is executed. Sometimes this is an adequate way for the program to be designed, but often there will be compelling reasons to perform certain lines out of order. We will discuss some of the reasons for doing this, and we will examine the keywords that make it possible, in the next section.



```
0/30
```

Figure 3.6: Investigating STEP

Controlling the Action

The keyword GOTO (located above the [G] key) can be used to tell the computer to perform a line out of sequence. GOTO is simply followed by the line numbers that you want the computer to perform next. The GOTO command can send control of the program forward to a line number that is larger than the current line:

```
90 GOTO 300
```

or it can send control backward, to a smaller line number:

```
90 GOTO 10
```

The GOTO command is a simple one, but the reasons for using it are often complicated. Generally, it is a good idea to use it sparingly in a program, because an excess of GOTOs can make the results of your program almost impossible to predict with certainty. But there are some situations when the use of GOTO cannot be avoided.

In both of the major programs of this chapter, we will use GOTO to create an "endless loop." The program, or a certain part of the program, will be repeated time after time; the program itself has no provision for ever ending. Your keyboard provides you with a means of interrupting such a program artificially. You can simply press the BREAK key, located in the lower-right corner of the keyboard.

The following lines are an example of an endless loop:

```
10 PRINT AT 10,13; "FLASH"
20 PRINT AT 10,13; "         "
30 GOTO 10
```

You can follow through the sequence of the program before you run it. First the PRINT statements at lines 10 and 20 are performed, then line 30 sends control of the program back to the beginning again. This process continues forever. Run the program to see the results. This is, of course, a rather frivolous example; subsequent programs will put GOTO to a better use.

Another BASIC command that instructs the computer to jump out of sequence to a new place in the program is GOSUB. This instruction means "go to a subroutine." Like the GOTO command, it is followed by a line number; for example:

```
GOSUB 200
```

In order to understand GOSUB, you have to know what subroutines are, and what they are used for. Often in a program that performs a number of different jobs, certain small and well-defined tasks may need to be performed time after time at different points in the program. When you set out to write such a program, you will recognize these tasks, and you will want a convenient and economical way to perform them whenever the need arises. BASIC lets you isolate such tasks into units called subroutines. Once you have written a subroutine, you can "call" it at any point in the program. It will do its job, and then return control of the program back to where the "call" originated.

Let's look at an example. Recall the short program we looked at that reads a horizontal and vertical address from the keyboard and then uses this address to place a pixel on the screen. This program will become a subroutine in a larger program we'll soon be writing. Here is what it will look like as a subroutine:

```
200 PRINT AT 21,0; "HORIZONTAL"
210 INPUT H
220 PRINT AT 21,0; "VERTICAL  "
230 INPUT V
240 PLOT H,V
250 RETURN
```

Notice that a line has been added: a RETURN instruction in line 250. RETURN is a keyword, located above the [Y] key. Every subroutine must have a RETURN; the instruction simply says that the job of the subroutine is complete. When the computer encounters the RETURN instruction, it automatically returns control of the program back to the place where the subroutine was called.

To call this subroutine, then, you can simply write the command:

```
GOSUB 200
```

We'll look at two short programs that use this subroutine. You should type both programs into your computer, along with the lines of the subroutine, and try them out.

Here is the first one:

```
10 PRINT "YOU MAY PLACE 5 PIXELS"
20 FOR I = 1 TO 5
30 GOSUB 200
40 NEXT I
50 STOP
```

This program allows you to place five pixels on the screen. It begins by printing a message at the top of the screen (line 10) and then enters a FOR loop. The GOSUB statement, at line 30, is part of the loop. Here is the sequence of events generated by this loop:

1. The FOR statement uses the control variable, I, to determine the number of repetitions.
2. The GOSUB statement sends control of the program down to line 200, where the job of inputting an address and placing a pixel is performed.
3. The last line of the subroutine (250) RETURNs control of the program back up to line 40, the NEXT statement.
4. NEXT increments the control variable, I, and the looping continues.

You can see from this sequence of events that the subroutine at line 200 will be called five times, allowing you to place five pixels on the screen; and then the program is complete. Line 50 is very important:

```
50 STOP
```

The keyword STOP (a shift character, located on the [A] key) is used to tell the computer that the program is over. In this case, the main body of the program is formed by lines 10 to 50. When the FOR loop has completed its five repetitions, you don't want the computer to continue on to line 200 to perform the subroutine again. So you have to instruct the computer explicitly to stop, which is the role of line 50. This is the first time we have had to use the keyword STOP. Up to now we have seen programs performed line by line, from the first line to the last line. When the computer performs the last line, it stops automatically. But in this program the final six lines of the program (lines 200 to 250) form a subroutine, which we want to perform only by means of the GOSUB command.

Here is the second example of a program that uses our subroutine:

```
10 PRINT "PLACE AS MANY PIXELS AS YOU WANT"
20 GOSUB 200
30 GOTO 20
```

This program lets you place as many pixels on the screen as you want. Lines 20 and 30 form an endless loop. Here is how the program works:

1. Line 20 calls the subroutine.
2. The subroutine does its job and then returns control to line 30.
3. The GOTO instruction in line 30 sends control back to line 20.

Run the program to see exactly what it does. You can continue entering pixel addresses as long as you want to. After reading each address, the computer will place a pixel on the screen. To stop the program, you'll have to enter the word STOP instead of typing an address. The BREAK key doesn't work when the computer is waiting to read input from the keyboard.

We have one last topic to discuss before we move on to the two major programs of this chapter. In the next section we will see how to use the keyword IF to instruct the computer to make decisions during the run of a program.

Decisions, Decisions . . .

You can often vastly increase the power and convenience of a program by including instructions that give the computer the power to "decide" between two different courses of action. You may have already encountered a problem in the pixel-placing program that could be solved by adding some decision lines to the program. First we'll look at the problem, then we'll see the solution.

Run the second pixel program again. You know that when you see the prompt

```
HORIZONTAL
```

you must enter a number that is less than or equal to 63. Likewise, when you see the prompt

```
VERTICAL
```

you cannot enter a number greater than 43. This is because the largest address that the PLOT instruction can handle is 63,43. Any address larger than that would fall somewhere outside of the screen area. Perhaps you have already seen what happens to the program if you accidentally enter an "illegal" address. If you haven't, then try it now. Enter the number 64 for the horizontal part of the address, and 44 for the vertical. Both of these numbers are larger than PLOT can handle. As a result, no pixel is placed on the screen; instead the program run is interrupted and the error message:

```
B/240
```

appears in the lower-left corner of the screen.

If you look at Appendix B you'll find the meaning of this error message. The error code "B" means that the number you have entered is outside the acceptable range of numbers. In this case, the range is defined by the computer for the keyword PLOT. Notice that the second part of the error message gives you the line number—240—where performance of the program stopped. If you look again at the subroutine you'll see that, sure enough, line 240 is the line that contains the PLOT command.

To avoid this problem, it would be very convenient if the computer could check the input values of the variables H and V before going on to perform the PLOT command. If either H or V were found to contain values too large for PLOT to handle, the ideal course of action would be to return to the INPUT statements for an acceptable value. This is exactly what we'll instruct the computer to do, using two IF statements.

A general description of the IF statement might look something like this:

```
IF (true-or-false statement) THEN (keyword command)
```

As you can see, the IF statement contains two parts. The first part begins with the keyword IF (located above the [U] key) and the second part begins with the word THEN (a shift character, located on the [3] key). After IF comes a statement that the computer evaluates to either true or false. This statement is usually in the form of an equality or an inequality. We will see in a moment how to write such statements. After the word THEN comes a command to the computer, expressed in the form of a keyword instruction.

The IF statement results in one of two possible courses of action:

1. If the true-or-false statement is true, then the computer performs the command expressed after THEN.
2. If the true-or-false statement is false, then the computer skips by the command after THEN and simply moves on to the next line of the program.

You can see how powerful the IF statement is. By this means, your programs can be made to respond accurately and efficiently to many different situations that occur during the run of a program.

Let's see exactly how to write an IF statement. The true-or-false statement involves learning how to use a new set of symbols on your keyboard. They are the equality and inequality symbols:

=	(is equal to)
<	(is less than)
>	(is greater than)
< =	(is less than or equal to)
> =	(is greater than or equal to)
< >	(is not equal to)

All of these symbols are shift characters, located on the [L], [N], [M], [R], [Y], and [T] keys, respectively. The symbols are used to compare two different values, which can be numerical or nonnumerical. Here are some examples of statements using these symbols:

```
I > 5
K < J
M <= 16
S$ = "YES"
H < > 15
```

All of the examples above compare either the values of two different variables, or the value of one variable and a literal value. All of these statements are either true or false. For example, in the first of these statements, if the variable I contains the value 4, the statement is false, because 4 is not greater than 5. If I contains the value

7, however, the statement is true, because 7 is greater than 5.

Here are examples of some IF statements that use these true-or-false statements:

```
IF I > 5 THEN PRINT "I IS TOO BIG"
IF K < J THEN INPUT J
IF M < =16 THEN GOTO 200
IF S$ = "YES" THEN STOP
IF H <> 15 THEN LET H = 100
```

Notice that in each case, THEN is followed by a keyword. All of the keywords on your keyboard can be used after THEN in an IF statement, although some of them are more commonly used than others. Keep in mind that the instruction after THEN is *only* performed if the true-or-false statement turns out to be *true*. Otherwise, the computer simply proceeds on to the next line of the program.

Now, for some meaningful examples of IF, let's return to the problem we outlined above. The solution to the problem requires a refinement of the subroutine that begins at line 200. When the computer reads each value from the keyboard, and assigns the values to the variables H and V, respectively, we would like to add a *test* to make sure both values are within the range that PLOT can handle. We can easily express this test in the form of an IF statement placed directly after each INPUT statement.

Here is the first one:

```
210 INPUT H
215 IF H > 63 THEN GOTO 210
```

Line 210 reads a value for H. Line 215 evaluates the statement H < 63 before deciding what to do next. If the variable H contains a value that is greater than 63, then the GOTO statement is performed. Control of the program goes back to line 210 for a new input value. If, on the other hand, the value of H is less than or equal to 63, then the GOTO command is skipped and the program proceeds normally to the next statement.

We can write a similar IF statement directly after the INPUT statement for V. This second IF, at line 235, evaluates the statement V > 43; it sends control back to line 230 if V > 43 is true.

The entire subroutine is shown in Figure 3.7. Study it carefully, and make sure you understand how the two IF statements work. Then try it out with one of the two short programs that we described above. In particular, when the program is running, try typing some incorrect values for the horizontal and vertical parts of the address. Now, instead of terminating the program, this kind of input error will simply result in rejection of the input value. For example, when you see the prompt:

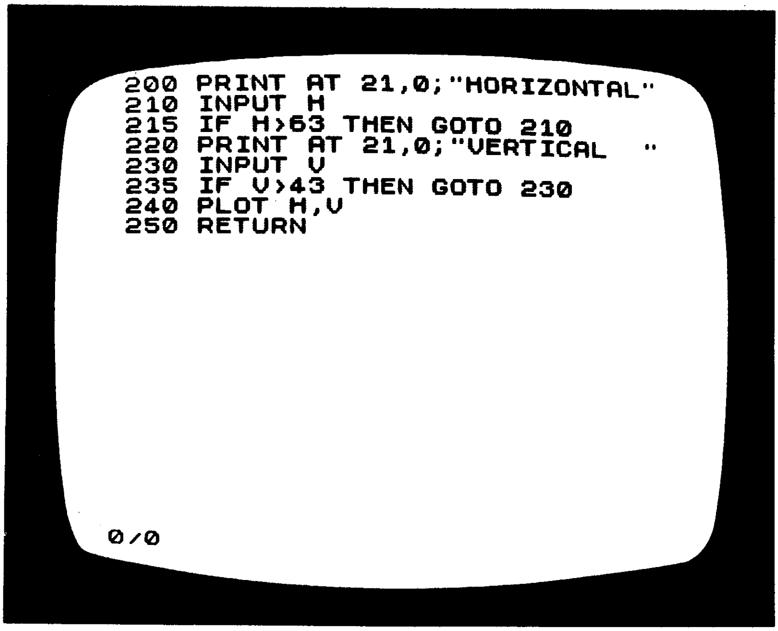
```
HORIZONTAL
```

try typing the number 66. When you do so, the prompt will remain the same, indicating that 66 will not work as the horizontal address. You have to type another value.

As a review of everything you've learned so far in this chapter, we'll now look at a graphics program that draws rectangles on the screen. We'll refer to it as "the rectangle program."

The Rectangle Program

The main body of the program is shown in Figure 3.8. The program uses the subroutine shown in Figure 3.7, so you'll have to include those lines when you type the program. Enter the entire



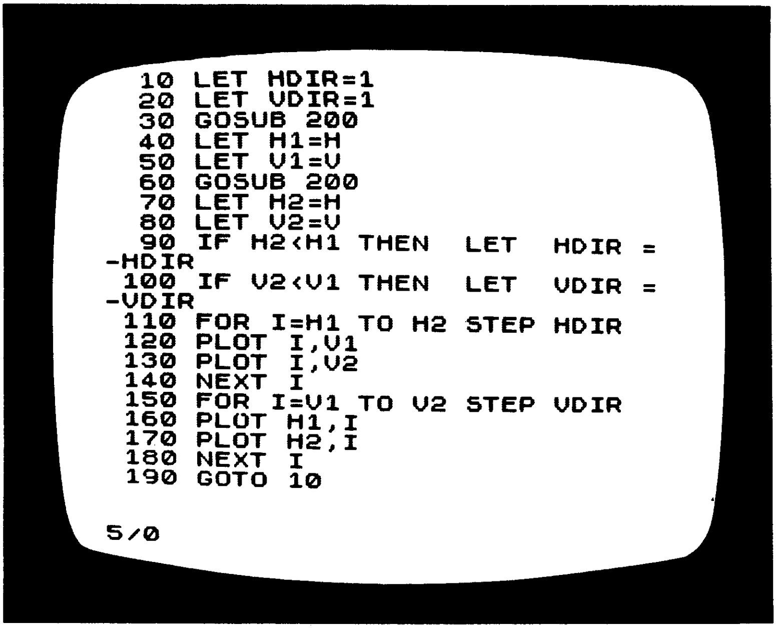
```
200 PRINT AT 21,0;"HORIZONTAL"
210 INPUT H
215 IF H>63 THEN GOTO 210
220 PRINT AT 21,0;"VERTICAL  "
230 INPUT V
235 IF V>43 THEN GOTO 230
240 PLOT H,V
250 RETURN




0/0
```

Figure 3.7: The Pixel Subroutine, Including IF Statements

program into your computer now and run it.

This program is simply an exercise designed to illustrate a number of programming tools in action. All the same, you can use it to produce some interesting images on the screen. Here is how it works: It begins by prompting you to enter two complete pixel addresses. In other words, you will see the HORIZONTAL and VERTICAL prompts twice, and the computer will wait for you to enter a number after each prompt. A pixel will appear on the screen as soon as you have entered a complete, valid address. When two pixels are determined, the computer will draw a rectangle on the screen, using the pixels as two diagonal corners of the rectangle. This process continues for as long as you want, allowing you to draw many rectangles of different sizes and shapes on the screen. Figure 3.9 shows a sample screen produced by this program. To end the program, enter the keyword STOP.

As you experiment with using the program, notice that it has the ability to draw the rectangles in different directions. To see what this means, begin by entering the addresses 5,5 and 25,25.

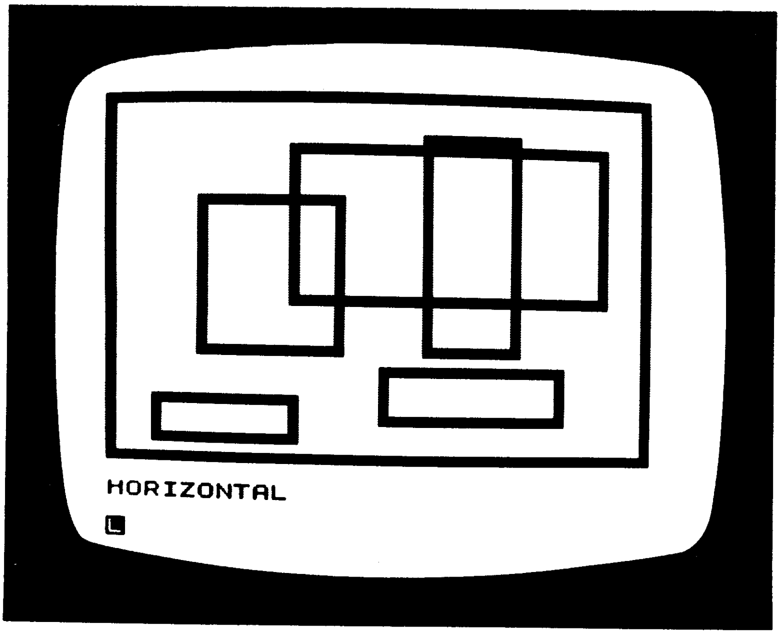


```
  10 LET HDIR=1
  20 LET VDIR=1
  30 GOSUB 200
  40 LET H1=H
  50 LET V1=V
  60 GOSUB 200
  70 LET H2=H
  80 LET V2=V
  90 IF H2<H1 THEN  LET  HDIR =
-HDIR
 100 IF V2<V1 THEN  LET  VDIR =
-VDIR
 110 FOR I=H1 TO H2 STEP HDIR
 120 PLOT I,V1
 130 PLOT I,V2
 140 NEXT I
 150 FOR I=V1 TO V2 STEP VDIR
 160 PLOT H1,I
 170 PLOT H2,I
 180 NEXT I
 190 GOTO 10

5/0
```

Figure 3.8: The Rectangle Program

Since the first address is lower than, and to the left of, the second address, the computer draws the top and bottom of the rectangle from left to right, and the sides of the rectangle from bottom to top. Now enter two more addresses: 45,43 and 25,23 in that order. In this case, the first address you entered is above and to the right of the second address, and as a result the rectangle is drawn in opposite directions from the first rectangle. The top and bottom are drawn from right to left; and the sides from top to bottom. This may seem like a small detail, but it is a carefully designed feature of the program. Let's examine the lines of the program. To look at the listing on your own screen, enter STOP to end the program run and then press ENTER again. The program is too long to be displayed all at once on the screen, but remember that you can use the keyword LIST to view different parts of the program. For example, if you wanted to see the subroutine at line 200, you would simply type:

```
LIST 200
```



```
HORIZONTAL
L
```

Figure 3.9: Results of the Rectangle Program

The first two lines of the program define the variables HDIR and VDIR:

```
10 LET HDIR = 1
20 LET VDIR = 1
```

The names of these variables stand for "horizontal direction" and "vertical direction," respectively. Throughout the run of the program, each of these variables will always contain either the value 1 or the value −1, and, as a result, they will be used to determine the direction in which the rectangle is drawn. We will see how as we look at subsequent lines of the program.

Lines 30 and 60 each contain calls to the subroutine at line 200:

```
GOSUB 200
```

Remember what this subroutine does:

1. It prints the prompt HORIZONTAL on the screen and then reads a value (not greater than 63) from the keyboard. It stores this value in the variable H.
2. It prints the prompt VERTICAL on the screen and then reads a value (not greater than 43) from the keyboard. It stores this value in the variable V.
3. It uses the PLOT command to place a pixel at the address H,V.

This subroutine must be called *twice* for each rectangle—to determine the two diagonal corners that will define the rectangle. Each time the subroutine is called, it will store new values in the variables H and V. The old values of H and V will be lost. But in order to draw the rectangle, the program has to keep track of the two distinct addresses of the two corners. This is the reason for lines 40, 50, 70, and 80 in the program. After each subroutine call, the program stores the new values of H and V in variables whose purpose is to "save" those values. Specifically, H1 and V1 save the values of the first address (i.e., from the first call to the subroutine); and H2 and V2 save the values of the second address (from the second call to the subroutine).

Study the form of the LET statements that assign values to the variables H1, V1, H2, and V2. For example, look at the first one:

```
40 LET H1 = H
```

This statement, paraphrased, says, "Take the current value of the variable H, and store that value in the variable H1." This is the first time we have seen a variable name on the right side of the equal sign in a LET statement. It's important for you to understand what the statement does—and what it *doesn't* do. It defines the variable H1 and gives it a value. But it does *not* affect the variable H in any way. After this LET command is performed, the variable H will still have the same value it had before.

Once these two addresses are determined, the computer must decide the directions in which to draw the rectangles. Lines 90 and 100 control this decision. Remember that HDIR and VDIR are both initially set to a value of 1. As we will see, this value indicates that the rectangle will be drawn from left to right and from bottom to top. The purpose of lines 90 and 100, then, is to *change* these directions, if necessary. If the second horizontal value, H2, is smaller than (i.e., to the left of) the first, H1, the rectangle must be drawn from right to left. This is indicated by changing the value of HDIR to −1:

```
90 IF H2 <  H1 THEN LET HDIR = -HDIR
```

Notice the LET command that appears after the word THEN. The variable HDIR appears on *both* sides of the equal sign. This LET statement says, "Take the *negative* of the current value stored in HDIR, and store that new value back in HDIR." In other words, since the original value of HDIR was 1, the new value will be −1. The original value is, of course, lost.

Next in the program there are two FOR loops. The first loop (lines 110 to 140) draws the top and the bottom lines of the rectangle, and the second loop (lines 150 to 180) draws its sides. As you study the way these loops are constructed, you'll understand the full significance of the direction variables, HDIR and VDIR.

Each loop contains two PLOT commands. We've already seen how two PLOTs within a FOR loop can produce two separate lines. In this case, they produce parallel lines. In the first loop, the control variable, I, is incremented from the value of H1 to the value of H2:

```
110 FOR I = H1 TO H2 STEP HDIR
```

The subsequent PLOT statements have fixed vertical addresses, at V1 and V2, respectively; they use the varying value of I as their horizontal addresses:

```
120 PLOT I,V1
130 PLOT I,V2
```

This is why you see the top and bottom lines of each rectangle drawn first when the program is running.

In the same way, the second FOR loop draws the sides of the rectangle by holding the horizontal addresses, H1 and H2, fixed, and varying the vertical addresses from V1 to V2:

```
150 FOR I = V1 TO V2 STEP VDIR
160 PLOT H1,I
170 PLOT H2,I
180 NEXT I
```

Now, what about the STEP clause of these two FOR statements? The easiest way to see why they are necessary is to think of entering two pixel addresses where the first address is above and to the right of the second address; for example:

address #1 : 45,43
address #2 : 25,23

Now substitute these values into the FOR statements and see what you have:

```
110 FOR I = 45 TO 25
150 FOR I = 43 TO 23
```

It turns out that these two FOR loops, as they appear above, would never perform their instructions even once, because the first number in the FOR statements is greater than the second number. In order to make these loops work, we have to add a STEP clause that will result in a *decrease* (or *decrement*) of the control variable, I, each time the loop is repeated. This is why the direction variables, HDIR and VDIR, must be set to −1 if the first address is greater than the second address. As a result, the FOR loops for the addresses above would be:

```
110 FOR I = 45 TO 25 STEP -1
```

```
150 FOR I = 45 TO 23 STEP -1
```

The STEP clause specifies that the control variable will be decreased by 1 each time the loop repeats.

If these details still seem a bit confusing, run the program again, and think about the two FOR loops as you watch the rectangles appear on the screen. Draw several rectangles, and observe the different directions in which they are drawn. You now know that these directions are determined by the order in which you enter the pixel addresses.

The last line of the main body of this program (that is, before the beginning of the subroutine) is line 190:

```
190 GOTO 10
```

This line creates an endless loop. You can keep drawing rectangles as long as you want to. The program itself has no provision for stopping, which is why you have to use the STOP command from the keyboard in order to end the program run.

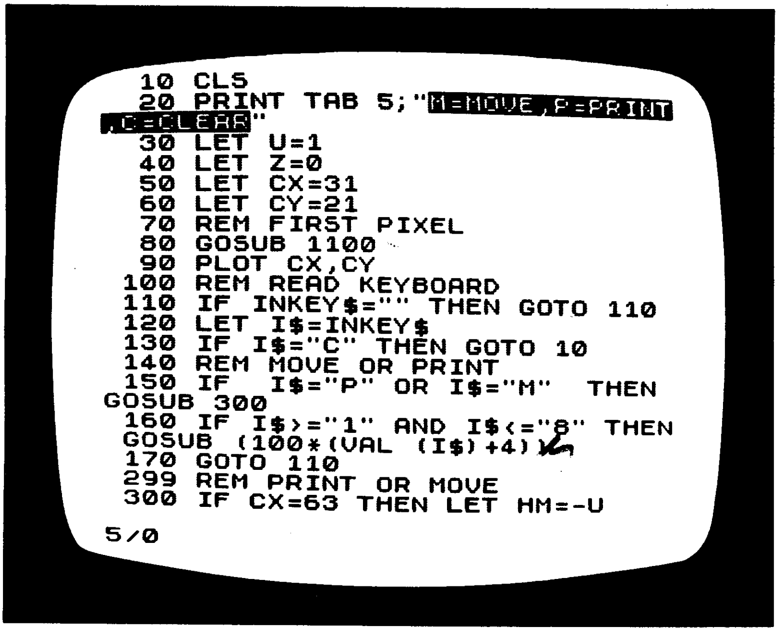
The final program of this chapter is also a graphics program, but one that is rather more interesting than the rectangle program. We will refer to this final program as "the picture program," because it is designed to help you draw pictures on the screen. We'll begin with instructions for using the program, so you can type it in and start enjoying it right away. (The program is long, so be sure you store a copy of it on cassette as soon as you've typed it all in.)

You'll also find a line-by-line explanation of how the program works. The purpose of including the program in this chapter is to deepen and expand your understanding of all the BASIC instructions you've learned about so far. The methods and techniques illustrated in the picture program are somewhat more advanced than anything you've seen so far. You may find that you'll have to live with the program for a while before you fully understand how it works. But don't rush yourself. You can have fun *using* the program even if you haven't quite cracked the shell of its inner logic.

THE PICTURE PROGRAM

The program listing is shown in Figures 3.10, 3.11, and 3.12. As you can see, the program is much too long to be displayed all at once on the screen. As you type the program into your computer, filling the screen with lines of instruction, you'll find that the computer continually prints and reprints the listing of the program on the screen. It always adjusts the portion of the program that appears on the screen so that the current line will be displayed. This process of adjustment can be rather slow when you're typing such a long program. To speed up the process, you can put the computer in fast mode. Simply enter the keyword FAST as an immediate command. If you do this, you'll find that the computer takes much less time adjusting the program listing. Once you have typed the program, and before you run it, return the computer to the slow mode by entering the keyword SLOW, again as an immediate command.

Check through all the lines of the program to make sure you have typed them correctly. Then enter the RUN command to run the program. Figure 3.13 shows the screen as it appears initially, when you first start the program. In the center of the screen is a single pixel, and some brief instructions are displayed at the top of the screen. You can move this pixel anywhere on the screen, in any of eight directions: up, down, left, or right; and the diagonal



```
  10 CLS
  20 PRINT TAB 5;"M=MOVE,P=PRINT
,C=CLEAR"
  30 LET U=1
  40 LET Z=0
  50 LET CX=31
  60 LET CY=21
  70 REM FIRST PIXEL
  80 GOSUB 1100
  90 PLOT CX,CY
 100 REM READ KEYBOARD
 110 IF INKEY$="" THEN GOTO 110
 120 LET I$=INKEY$
 130 IF I$="C" THEN GOTO 10
 140 REM MOVE OR PRINT
 150 IF  I$="P" OR I$="M"  THEN
GOSUB 300
 160 IF I$>="1" AND I$<="8" THEN
 GOSUB (100*(VAL (I$)+4))
 170 GOTO 110
 299 REM PRINT OR MOVE
 300 IF CX=63 THEN LET HM=-U

5/0
```

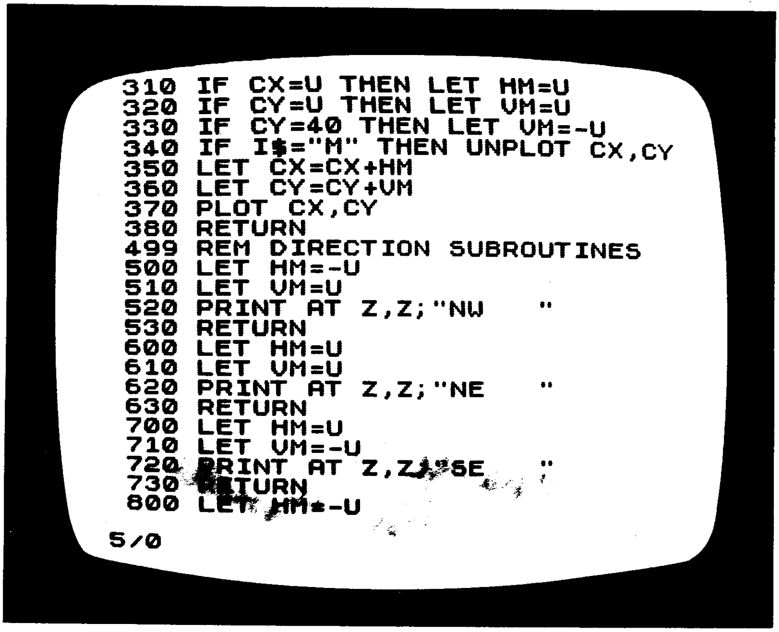
Figure 3.10: The Picture Program

directions, described as NW, NE, SE, SW. The word in the upper-left corner of the screen always describes the current direction. (At the beginning of the program the direction is "UP.") You can change the direction by pressing one of the number keys, from 1 to 8. For the diagonal directions, press keys 1 to 4. You can read the directions from the graphics characters on these keys:

1 :	NW
2 :	NE
3 :	SE
4 :	SW

The other directions are indicated by the arrows shown on keys 5 to 8:

5 :	left
6 :	down
7 :	up
8 :	right



```
310 IF CX=U THEN LET HM=U
320 IF CY=U THEN LET VM=U
330 IF CY=40 THEN LET VM=-U
340 IF I$="M" THEN UNPLOT CX,CY
350 LET CX=CX+HM
360 LET CY=CY+VM
370 PLOT CX,CY
380 RETURN
499 REM DIRECTION SUBROUTINES
500 LET HM=-U
510 LET VM=U
520 PRINT AT Z,Z;"NW     "
530 RETURN
600 LET HM=U
610 LET VM=U
620 PRINT AT Z,Z;"NE     "
630 RETURN
700 LET HM=U
710 LET VM=-U
720 PRINT AT Z,Z;"SE     "
730 RETURN
800 LET HM=-U

5/0
```

Figure 3.11: The Picture Program (cont.)

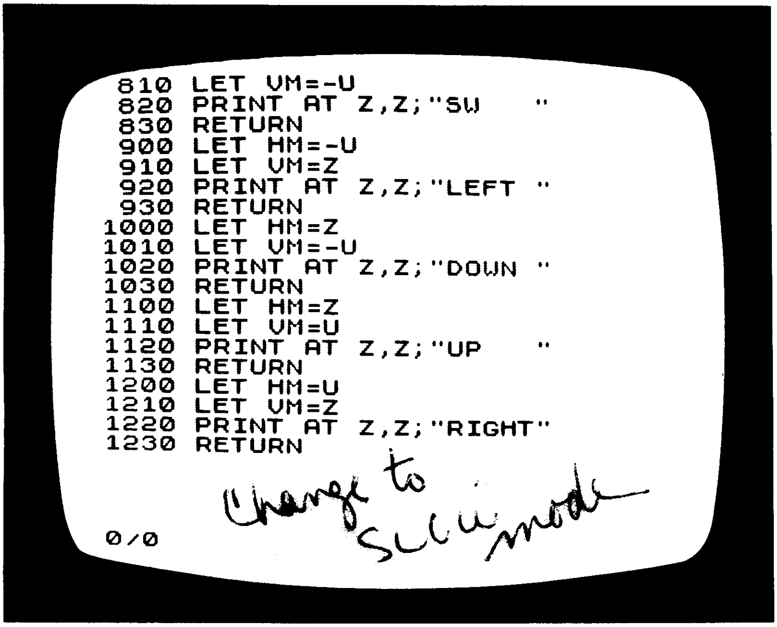
Try pressing all eight of these keys, one at a time. Each time you press a key, the word in the upper-left corner of the screen will change.

Once you have selected the direction in which you want to move, you can move the pixel by pressing one of two keys. The [M] key simply moves the pixel. Press the key once to move it one address in the indicated direction, or hold your finger down on the [M] key for continual movement.

The [P] key (for "print") leaves a trail of pixels behind when the pixel moves. Again, you can either press the key for one move at a time, or you can hold the key down for a number of moves. Take a moment to experiment with both of these keys in several different directions now.

You can erase any pixel on the screen by moving backwards over it. Press the correct direction key, toward the pixel you want to erase, then press the [M] key.

If you want to start over with a fresh screen, press the [C] key to "clear" the current drawing away.



```
 810 LET VM=-U
 820 PRINT AT Z,Z;"SW    "
 830 RETURN
 900 LET HM=-U
 910 LET VM=Z
 920 PRINT AT Z,Z;"LEFT  "
 930 RETURN
1000 LET HM=Z
1010 LET VM=-U
1020 PRINT AT Z,Z;"DOWN  "
1030 RETURN
1100 LET HM=Z
1110 LET VM=U
1120 PRINT AT Z,Z;"UP    "
1130 RETURN
1200 LET HM=U
1210 LET VM=Z
1220 PRINT AT Z,Z;"RIGHT"
1230 RETURN

0/0
```

Change to slow mode

Figure 3.12: The Picture Program (cont.)

When the moving pixel reaches one of the sides of the screen it will automatically be "reflected" back into the screen. If it is moving in a diagonal direction, the angle of reflection will be 90°. If it is moving up, down, right, or left, it will be reflected in the opposite direction.

You'll find that the longer you play with this program, the better the results will become. You can use it to draw almost anything on your screen. The main limitation, of course, is one of resolution, determined by the size of the pixel. Figure 3.14 shows a sample picture drawn using this program.

Inside the Program

Type the BREAK key to interrupt the program; then type ENTER to list the program line on the screen. This program is organized into a main controlling section, and a number of subroutines. We'll first see how the main program section works together with the subroutines; then we'll look in detail at some of the lines of the program.

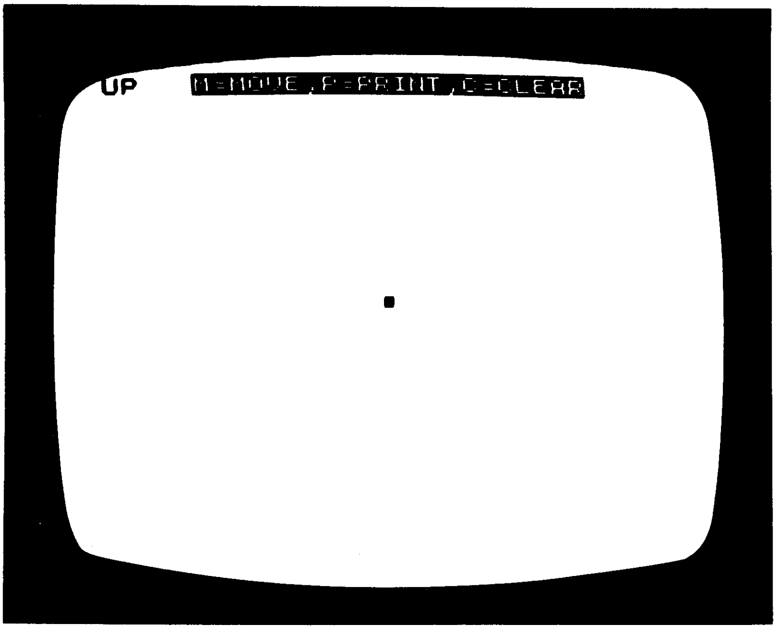
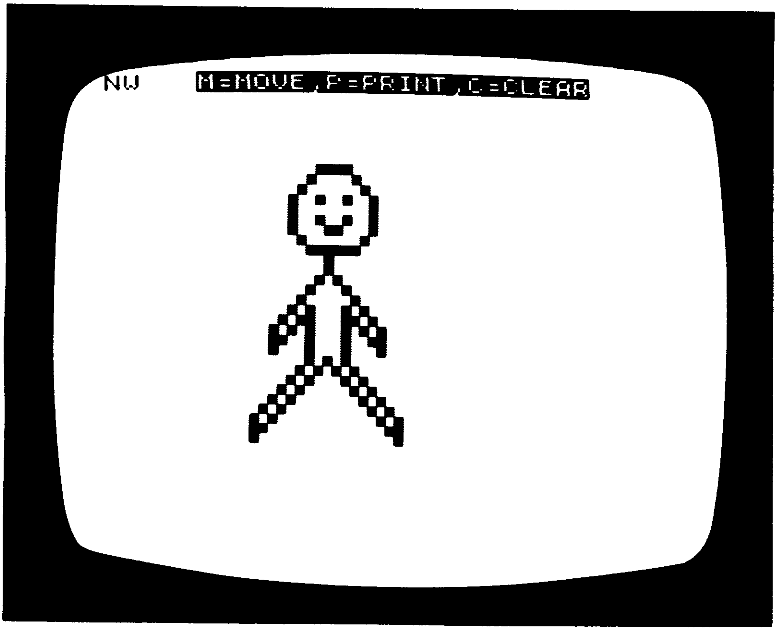


Figure 3.13: Starting the Picture Program

This program demonstrates the tremendous power of subroutines. The main tasks of this program are organized into their own individual subroutines. The result is that the program is easy to understand, and easy to correct or revise. Once you know the task of each subroutine, you'll know exactly where to look if you want to change some detail of the program.

Briefly, here is how the program is organized. Eight short subroutines at the end of the program are charged with the task of changing the direction of movement. Each time you press one of the direction keys, one of these subroutines is called. These subroutines control the values of two variables, named HM (for "horizontal movement") and VM (for "vertical movement"). These two variables, in turn, determine the direction the pixel will move in when you press the [P] key or the [M] key. The variables HM and VM always have one of three values: 1, 0 or −1. Besides controlling these variables, the direction subroutines print the current direction in the upper-left corner of the screen. Each of these subroutines is only four lines long, and each one begins on a line number



```
NW     M=MOVE,P=PRINT,C=CLEAR
```

Figure 3.14: A Sample Picture

that is a multiple of 100 (500, 600, 700 . . .). This is a very significant fact for the GOSUB statement that calls these subroutines, as we will soon see.

The subroutine that moves the pixel is located at line 300. It is longer and more complicated than the direction subroutines. First it must test to see if the current position of the moving pixel is at one of the four edges of the screen. If it is, the subroutine must adjust one of the variables HM or VM to reverse the direction of movement. Then the subroutine must determine whether the pixel is just moving, or whether it is leaving a trail behind it (i.e., whether you are pressing the [M] key or the [P] key). If the pixel is just moving, the previous position on the screen must be erased (with the UNPLOT command). Finally, this subroutine determines the new address of the pixel, by adding HM and VM to the current address, and PLOTs the new position.

Lines 110 to 170 read the keyboard and call the correct subroutines, depending on which key you have pressed. These lines form an endless loop, as you can see from line 170:

```
170 GOTO 110
```

The result is that the computer spends much of its time simply waiting for you to press a key on the keyboard. In this program, unlike other programs you have seen up to now, the keyword INPUT is not used to read the keyboard. Instead, we use the programming function called INKEY$ (located below the [B] key on your keyboard). We'll see how INKEY$ differs from INPUT when we discuss these lines in detail.

Finally, since we've moved our way backwards through the program, lines 10 to 90 are what we might call the "initialization" section of the program. Line 10 clears the screen (with the CLS command). Line 20 prints the reverse-video instruction line at the top of the screen, making use of the TAB function to position the message five spaces forward from the left edge of the screen. Lines 30 to 60 define (or *initialize,* we might say) several very important variables; we might as well begin our detailed discussion of the program with these variables.

Lines 30 and 40 define the variables U (for "unity") and Z (for "zero"):

```
30 LET U = 1
```

```
40 LET Z = 0
```

The reason for these variables is a rather long story. It has more to do with the nature of your computer than with this particular program. As you know, your computer has a limited amount of memory. This program is designed to fit into—and operate with—the computer's original 2K of memory. When you start writing relatively long programs like this one—especially programs that make heavy use of the TV screen when they are performed—you will begin looking for ways to conserve memory space. Don't forget that the computer has to use its memory for several different tasks when you are running a program. Part of the memory is reserved for storing the program instructions themselves; another part is required for keeping track of the information that is being sent to the screen; and still more memory is needed for other, less visible, tasks. One obvious way that you can conserve memory, then, is to use as little of it as possible for the program itself. In the picture program, that's where the variables U and Z come in.

The computer needs much less memory to store a character than it does a number. The computer has a special system for storing numbers; this system (called *floating-point binary,* in case you want to investigate further) is designed to store numbers as accurately as possible in a relatively small amount of memory. Accuracy, of course, becomes important to you when you start using your computer for calculations. This system takes up several times more memory for each literal number than is required for a character. So each time a literal number appears in your program listing, you should ask yourself if there is some way to replace that number with a letter.

In the picture program, the numbers 0 and 1 are used many times. Knowing this, we can perform a neat little trick to save a lot of memory space. We can assign the values 0 and 1 to two variables at the beginning of the program, and then use those variables throughout the program to represent the values 0 and 1. This is exactly what lines 30 and 40 do. As a result, when you read this program, any time you see the variable names Z and U, you should think of them as the "constant" values 0 and 1. This may make the program slightly harder for you to decipher, but it's worth it in the end. After all, your memory capacity is considerably larger than the computer's.

The other two variables defined near the beginning of the program will not take nearly as long to explain. The variables CX and CY represent the current horizontal position and the current vertical position, respectively. (If you are mathematically oriented, you know that the letters X and Y are traditionally used as variable names for the horizontal and vertical positions on a graph.) Lines 50 and 60 initialize these two variables:

```
50 LET CX = 31
60 LET CY = 21
```

This means that the pixel you see at the approximate center of the screen when the program begins is at the address 31,21.

Line 70 is the first of several REM lines that appear in this program. The keyword REM stands for remark. (It is located above the [E] key on your keyboard.) REM lines do not result in any action by the computer; during the program run, the computer ignores them. The usefulness of REM lies in the opportunity it gives you to "document" your program within the program listing itself. After the keyword REM you can write anything you want. Usually you'll write a few short words—your own words, not BASIC words—that describe what is happening at a certain point in the program. Then, each time you look at your program listing, the REM lines will be there to help you understand what the program does.

Unfortunately, REM lines take up room in the computer's memory. For this reason, you should use them sparingly. In addition, if you find yourself running short of computer memory while you are developing a program, you can always begin deleting (or condensing) any REM lines that you have written. But in general you'll find that a well-placed, well-written comment, in the form of a REM line, can save you lots of time when you are trying to revise, correct, or simply understand a program.

The initialization section of the program has two more lines. Line 80 initializes the direction of movement by calling one of the direction subroutines:

```
80 GOSUB 1100
```

The subroutine at line 1100 is the one that sets the direction to "UP."

Finally, line 90 plots the initial pixel:

```
90 PLOT CX,CY
```

This is the pixel that is at the center of the screen at the beginning of the program run.

The initialization section of the program is performed once when the program begins, and again whenever you press the [C] key to clear the screen. Line 130 uses a GOTO statement to "reinitialize" the program for an input character of "C":

```
130 IF I$ = "C" THEN GOTO 10
```

Line 110 reads the keyboard, using the INKEY$ command. Each time INKEY$ is performed, the computer "scans" the keyboard once to see if you are pressing a key. If you are pressing exactly one key, INKEY$ returns the character represented by that key—a letter "A" to "Z," or a digit "0" to "9." Notice that INKEY$ treats the digits as characters, not as numbers. This is an important distinction. It means that the computer is *not* storing these particular digits as numbers to be used for calculations (that is, *not* in the "floating-point binary" form) but rather as simple characters, like any others on the keyboard.

Since we want the computer to pay constant attention to keys pressed on the keyboard, we have to write the INKEY$ command within a loop:

```
110 IF INKEY$ = "" THEN GOTO 110
```

The GOTO statement, after THEN, sends control of the program right back to the beginning of the same line! This line, paraphrased, says, "If INKEY$ returns no character at all (meaning that no key is being pressed on the keyboard), go back to the beginning of the line and perform INKEY$ again." As long as you're not pressing a key, the computer is stuck performing line 110 time after time. But as soon as you do press a key, the true-or-false statement:

```
INKEY$ = ""
```

becomes false, and control of the program proceeds to the next line.

Since you have already run this program, you know that INKEY$ is completely different from INPUT in the action that it causes. INPUT prints each character that you type in the lower-left corner of the screen. INPUT also waits for you to press the EN-

TER key before it actually stores your input data in a variable. INKEY$ does neither of these things; it simply reads a single character from the keyboard and returns that character to the program.

Once INKEY$ has returned a character, line 120 stores the character in the string variable I$:

```
120 LET I$ = INKEY$
```

Then, lines 130, 150, and 160 test the value of I$—in three different IF statements—to decide what to do next. We've already seen that line 130 sends control of the program back to the beginning, to clear the screen.

Lines 150 and 160 illustrate a feature of the IF statement that we haven't discussed yet—the use of the words AND and OR in true-or-false statements. These two words allow you to test for more than one equality or inequality in a single IF statement.

Line 150, which tests for the P or M keys, contains the word OR:

```
150 IF I$ = "P" OR I$ = "M" THEN GOSUB 300
```

In other words, if the variable I$ contains either the character P *or* the character M, then line 150 calls the subroutine at line 300. A statement containing the word OR is true if *either* or *both* of the elements are true.

Line 160 is the most difficult line in the program. It tests to see if you are pressing one of the direction keys, "1" to "8":

```
160 IF I$> = "1" AND I$< = "8" THEN
                    GOSUB (100 * (VAL (I$) + 4))
```

A statement containing the word AND is true only if *both* of the elements are true. Thus, this line calls a subroutine only if the value of I$ is between the character "1" and the character "8". The GOSUB statement in line 160 is different from other GOSUBs we've seen in that it *calculates* the line number of the subroutine it calls. We will see that this gives line 160 the ability to call any one of eight different subroutines—a very powerful feature, indeed. The key to this feature lies in the expression after the word GOSUB:

```
(100 * (VAL (I$) + 4))
```

There are some things about this expression that you will understand better after you've read Chapter 4. (In fact, line 160 should continue to provide you with food for thought for a long time. Don't worry too much if you don't understand it all right away.) Briefly, this expression produces a multiple of 100 between 500 and 1200; you'll recall that these are the line numbers of the direction subroutines. The line numbers are calculated in three steps:

1. The function VAL converts the value of I$ from a character (between "1" and "8") to a number (between 1 and 8).
2. The expression adds 4 to this number, yielding a number between 5 and 12.
3. The result of step 2 is then multiplied by 100, giving a number between 500 and 1200.

The important thing to realize is that you can write a single GOSUB statement that will call one of several different subroutines. The statement can "decide" which subroutine to call, based on a calculation that includes a variable.

To summarize, lines 130, 150, and 160 can result in a subroutine call, but only if you are pressing one of the keys that has meaning to this program—C, M, P, or 1 to 8. If you press any other key, none of the IF statements will result in a subroutine call, and the program will simply continue on to line 170:

```
170 GOTO 110
```

Thus, one way or another, the program always returns to the INKEY$ statement in line 110.

Now let's look briefly at the subroutines, beginning with the direction subroutines. These short subroutines all have the same format. Each one has two LET statements that assign values to the variables HM and VM. In the subroutine that moves the pixel, the values of these two variables are used to adjust the current address in the correct direction. To see how this works, look at the "NW" direction subroutine, at line 500. It sets the horizontal movement to −1 and the vertical movement to 1:

```
500 LET HM = -U
510 LET VM = U
```

You can see that subtracting 1 from the horizontal part of a pixel address will move the pixel to the left (i.e., "west"). Likewise, adding 1 to the vertical part of the address will move the pixel up (i.e., "north").

Each of the direction subroutines ends by printing the new direction in the upper-left corner of the screen (at address 0,0):

```
520 PRINT AT Z,Z; "NW  "
```

In lines 350 and 360 of the pixel-moving subroutine, the current address (CX,CY) is changed by the values of HM and VM:

```
350 LET CX = CX + HM
360 LET CY = CY + VM
```

Line 350 can be paraphrased as "add the value of HM to the value of CX, and store the result in the variable CX." The old value of CX is lost, as is the old value of CY in line 360.

Once the new address has been calculated, the pixel can be placed on the screen:

```
370 PLOT CX,CY
```

The subroutine at line 300 has two more features you should study. Lines 300 to 330 test the current address to see if the pixel has arrived at one of the four edges of the screen. If it has, the value of HM or of VM must be changed to move the pixel back into the screen area. For example, if the horizontal address reaches 63 (the right side of the screen), HM, the horizontal movement variable, must be changed to −1 to move the pixel to the left:

```
300 IF CX = 63 THEN LET HM = -U
```

Finally, in line 340, the pixel-moving subroutine tests to see if it should erase the current pixel before placing the new pixel. If you pressed the [M] key, UNPLOT erases the pixel at CX,CY:

```
340 IF I$ = "M" THEN UNPLOT CX,CY
```

If you pressed the [P] key, the UNPLOT command is not performed, and the current pixel stays on the screen along with the next one.

By studying this program carefully you can refine your understanding of several BASIC statements, including LET, IF, and GOSUB. In addition, this program introduces you to several new words in the BASIC vocabulary—REM, INKEY$, TAB, AND, OR, and VAL. If all this was too much to absorb in one reading, you should come back to this program later, perhaps after you have read the final chapters of this book.

SUMMARY

We've covered a lot of ground in this chapter. The following list will help you to recall each of the BASIC commands you have learned and to review what they do:

- defining variables and giving them values—LET, INPUT
- reading information from the keyboard—INPUT, INKEY$
- displaying information on the screen—PRINT, AT, TAB, CLS
- displaying graphics on the screen—PLOT, UNPLOT
- repeating instructions, creating loops—FOR, TO, STEP, NEXT, GOTO
- controlling the order of performance—GOTO, GOSUB, RETURN
- making decisions—IF, AND, OR, THEN
- documenting your program—REM.

Chapter 4

Take Five: Numbers on Your Computer

INTRODUCTION

In this chapter, you will learn how to use your computer to perform numerical calculations. The chapter begins with a discussion of the arithmetic operations, followed by a brief survey of some of the more common arithmetic functions. You'll see several examples of all these features in action.

You'll also find two important and useful programs presented in this chapter. The first program turns your computer into a "super calculator," complete with operations, functions and memory. The second program, the longest program in this book, will allow you to create bar graphs for any kind of numerical data. You'll learn how to use both of these programs, and you'll have the opportunity to expand your understanding of BASIC programming by studying the structure and logic of the programs. In particular, you'll learn about *arrays* in the second program. An array is an essential structure in BASIC programming that allows you to store many items of data under a single variable name.

CALCULATIONS ON YOUR COMPUTER

The arithmetic operations that your computer can perform are addition, subtraction, multiplication, division, and exponentiation. The symbols used for these operations are:

+	addition
–	subtraction
*	multiplication

/	division
**	exponentiation

One way you can instruct the computer to perform a calculation is in a PRINT statement. The computer will begin by evaluating the calculation, and then will display the results on the screen. To see how this works, enter each of the following as immediate commands:

```
PRINT 15.193 + 6.5
PRINT 1235 - 872
PRINT 18 * 97
PRINT 183 / 5
PRINT 9 ** 3
```

Each time you enter one of these commands, you will see the result of the calculation, almost immediately, displayed at the top of the screen. The last of these five PRINT commands is an example of exponentiation. The expression "9 ** 3" means 9 to the power of 3, or 9 cubed. The result, as you can see, is 9 × 9 × 9, or 729. (The "**" symbol is a shift character located on the [H] key.)

You can also include multiple arithmetic operations in a single statement. For example, the following statement results in the sum of two products:

```
PRINT 9 * 8 + 5 * 3
```

The number displayed on the screen is 87, the sum of 72 and 15. Notice that the computer first evaluated both of the multiplication operations, then added the two products together. This is an essential point. When the computer evaluates multiple operations, it does so in a fixed order. You must always be aware of this order of operations, or risk receiving erroneous results.

The order of operations is as follows:

1. exponentiation, from left to right;
2. multiplication and division, from left to right;
3. addition and subtraction, from left to right.

To explore this fixed order, enter the following line:

```
PRINT 1 + 2 ** 3 * 4 - 5
```

Could you have predicted the correct result, 28? Here is how the arithmetic statement is evaluated: First, the exponentiation, 2 ** 3, which equals 8; then the product, 8 * 4, resulting in 32; finally, the addition of 1 and subtraction of 5, giving 28.

Obviously, if you had entered this line expecting the computer to perform the calculations from left to right—no matter what the nature of the operations—you would have been very surprised. Performing each operation in a left-to-right sequence, the result would be 103. (Try it: 1 plus 2 is 3; 3 cubed is 27; 27 times 4 is 108; 108 minus 5 is 103.) Clearly, you must always think carefully about the computer's order of operations whenever you formulate an arithmetic calculation.

Often, however, you will want to perform calculations that do not conform conveniently to this fixed order. Fortunately, BASIC gives you a way to override the normal order and establish your own order. To do this, you must use parentheses in arithmetic expressions.

When part of an expression is enclosed within parentheses, that part is evaluated first. For example, enter these two statements, one at a time:

```
PRINT 1 + 2 * 3
PRINT (1 + 2) * 3
```

In the first statement, the multiplication is performed first, and then the addition, yielding 7. In the second statement, the computer first evaluates the sum that is enclosed in parentheses, then multiplies the sum by 3; the result is 9.

You can also write statements that have levels of parentheses. (Parentheses inside other parentheses are sometimes called *nested* parentheses.) The computer deals with the *innermost* parentheses first, then works on to the outermost. The following statement, then, with parentheses, would be evaluated from left to right:

```
PRINT (((1 + 2) ** 3) * 4) - 5
```

The result of this statement, as we've already discussed, will be 103. Notice an important fact about nested parentheses: Each open-parenthesis symbol "(" must be matched by a close-parenthesis symbol ")". If you accidently write an expression that contains unmatched parentheses, the computer will refuse to perform the calculation.

We've already seen that arithmetic expressions can contain variables; the expression will be valid as long as the variables it contains are already defined. This applies to LET statements, PRINT statements, and to most other BASIC statements where numerical values are called for. To see for yourself that this is true, enter the following sequence of instructions as immediate commands:

```
LET A = 1
LET B = A + 2
LET C = B ** 3
LET D = C * 4
LET E = D - 5
PRINT A, B, C, D, E
```

Each LET statement in this sequence uses the value of the variable defined in the previous statement. The PRINT command then displays the values of all the variables. Notice that the final variable, E, contains the value 103, the result of the expression (((1 + 2) ** 3) * 4) – 5. The other variables contain the values of the intermediate calculations.

The PRINT statement above illustrates an additional feature of BASIC that we haven't discussed yet: the use of *commas* to separate elements of the display. A comma tells the computer to space the elements apart by half a screen width across. Thus, the result of this PRINT statement is the following display:

```
1                3
27               108
103
```

The difference between a semicolon and a comma in a PRINT statement is very important. The semicolon, you'll recall, instructs the computer to display elements side-by-side.

You should be aware of one further point involving calculations: Division by zero is not defined for the computer; writing an instruction that results in division by zero will end in an error message. Knowing this, you would not intentionally write a statement such as:

```
LET E = 1 / 0
```

But problems may often arise when you write division statements that contain variables in the denominator; for example:

```
LET E = 1 / D
```

If the variable D should ever contain the value zero when this LET statement is performed, the computer will give you an error message. This can be a particularly difficult problem in a program in which you're not sure in advance what values D will take. One way of avoiding the problem is to write the LET statement as a clause in an IF statement:

```
IF D < > 0 THEN LET E = 1 / D
```

This statement insures that the LET instruction will only be performed if the variable D does not contain a value of zero.

You've seen that your computer's keyboard offers you several useful mathematical functions. You can type them by shifting the keyboard into the function mode. Let's explore the use of these functions in calculations.

ARITHMETIC FUNCTIONS

We'll look at the following functions:

- ABS (for absolute value; located below the [G] key);
- INT (for integral value; located below the [R] key);
- SGN (for sign; located below the [F] key);
- SQR (for square root; located below the [H] key).

These are functions you might well find yourself using, even if you are not mathematically inclined. Your computer also offers other sets of functions, for specialized uses. These include the trigonometric functions (SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN) and the exponential functions (EXP, LN). If you are interested in these functions, you can read about them in Appendix A.

The absolute value function, ABS, supplies the *positive* value of any number. In other words, if a number is negative, ABS makes it positive. If a number is positive, ABS does not change it. Enter the following two commands to see how ABS works.

```
PRINT ABS -45
PRINT ABS 45
```

The result of both commands is 45. Notice that the minus sign in the first statement simply expresses the sign of 45, *not* an operation between two different numbers.

The INT function eliminates the fractional part (if any) of a decimal number. For example, try these commands:

```
PRINT INT 19.34
PRINT INT 19.81
```

The result of both the statements is 19. Notice that INT does not round up; it simply *truncates* to the next lower integer. Later in this chapter we will see a way to use INT to create a rounding function.

The function SGN always results in one of three values, depending on the sign of the number you write after it:

−1	if the number is negative
0	if the number is zero
1	if the number is positive.

The statement:

```
PRINT SGN -45, SGN 0, SGN 45
```

produces the sequence −1, 0, and 1.

Finally, SQR gives the square root of a positive number. For example:

```
PRINT SQR 81
```

results in 9. If you type a negative number after SQR, the computer will give you an error message; the square root of a negative number is not defined.

Functions can be part of statements that include more than one arithmetic operation; but, again, you'll have to pay attention to the order of operations. Functions are evaluated before any of the five arithmetic operations (unless those operations are enclosed in parentheses). For example, enter the statement:

```
PRINT 10 * SQR 81 + 19
```

The result is 109. The computer first finds the square root of 81, giving 9; then it multiplies 9 by 10, giving 90; finally it adds 19, giving 109. Compare this to the statement:

```
PRINT 10 * SQR (81 + 19)
```

This statement yields 100. The computer first performs the addition located between parentheses, then takes the square root, then performs the multiplication.

Finally, functions can work on the results of other functions. In other words, you can use functions in complex expressions such as:

```
SQR INT 9.73
```

This expressions says, "find the square root of the integral value of 9.73." If you enter this expression into your computer as part of a PRINT statement, you'll see that the result is 3.

In the next section we'll learn more about using the INPUT statement for calculations, and we'll run a short program that will help us experiment with this feature.

CALCULATIONS WITH THE INPUT STATEMENT

Type NEW to clear any program lines out of your computer's memory; then enter the following three-line program:

```
10 INPUT V
20 PRINT V
30 GOTO 10
```

On the surface this doesn't look like a very useful program. Line 10 reads a numeric value from the keyboard, and stores the value in the variable V. Line 20 displays the value of V on the screen. Line 30 sends control of the program back up to line 10, forming an endless loop. When you run the program, the computer will wait for you to enter a number from the keyboard, and then will display the number on the TV screen.

As we will see, the computer can actually read arithmetic expressions from the keyboard during the run of a program. This is a unique, and very useful, feature of the T/S 1000's BASIC. Instead of merely typing a number at the keyboard, you can type any valid arithmetic expression, including any of the five arithmetic operations, and any numerical functions. The computer will immediately perform the calculations specified in the expression and then will store the *result* of the calculations in the INPUT variable, in this case V.

Run the program now and try it out. Begin by entering a simple number, say 10; the number will appear at the top of the screen. You can see that the program is working.

Next enter an expression. Try this one:

```
10 * SQR 49 + 16 / 8
```

The number 72 will be displayed at the top of the screen. The computer has evaluated the expression, storing the result, 72, in the variable V.

Now enter the expression:

```
V / 9
```

The computer will respond by displaying the number 8 on the screen. Your expression V / 9 meant, "divide the current value of the variable V by 9." Since V held the value 72 from the previous calculation, the result of the division is 8.

To summarize, then, we have seen that the computer will accept any valid numeric expression in response to a numeric INPUT statement. The expression may even include variables, as long as the variables have already been defined.

We're now ready to look at the program that turns your computer into a super calculator.

BUILDING A SUPER CALCULATOR

The super-calculator program appears in Figure 4.1. Type the program carefully into your computer, and then run it. You'll see the message:

```
SUPER CALCULATOR
```

in reverse video, and the L cue, between quotes, indicating that the computer is waiting for a string input.

You can use this program to find the answer to any calculation problem. The expression you type into the computer may include:

- all five arithmetic operations;
- any of the mathematical functions on your keyboard;
- complex expressions, with nested parentheses and nested functions;
- two variables: M1 and M2, representing the program's

memory function. (We will discuss this feature shortly.)

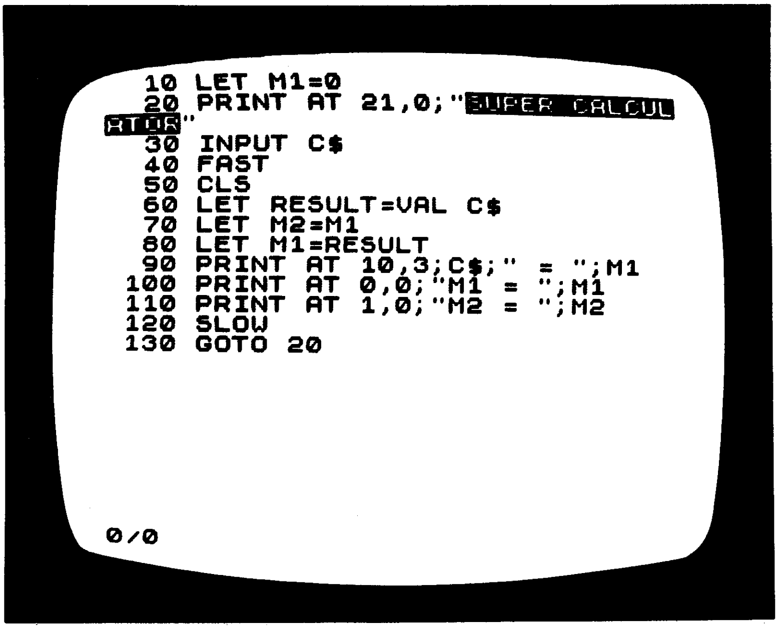
In other words, you can enter any calculation that the computer will accept as a valid numerical expression. The program is in the form of an endless loop; you can use it to perform as many calculations as you want.

Let's give it a try. Type the following expression at the keyboard:

```
1 + INT (SQR (2 * 3) * 4)
```

Press the ENTER key and watch the action. The screen goes blank very briefly. Then, when the information returns, your screen should look like Figure 4.2. Study the screen carefully to find out everything it has to tell you.

First, the expression that you entered into the computer is "echoed" for you in the center of the screen. This is important; how often have you performed a long series of calculations on a pocket calculator, and then, when the result appeared, wondered nervously if you'd pressed all the correct keys? Input errors are probably the most common cause of incorrect results when we use



```
  10 LET M1=0
  20 PRINT AT 21,0;"SUPER CALCUL
ATOR"
  30 INPUT C$
  40 FAST
  50 CLS
  60 LET RESULT=VAL C$
  70 LET M2=M1
  80 LET M1=RESULT
  90 PRINT AT 10,3;C$;" = ";M1
 100 PRINT AT 0,0;"M1 = ";M1
 110 PRINT AT 1,0;"M2 = ";M2
 120 SLOW
 130 GOTO 20




0/0
```

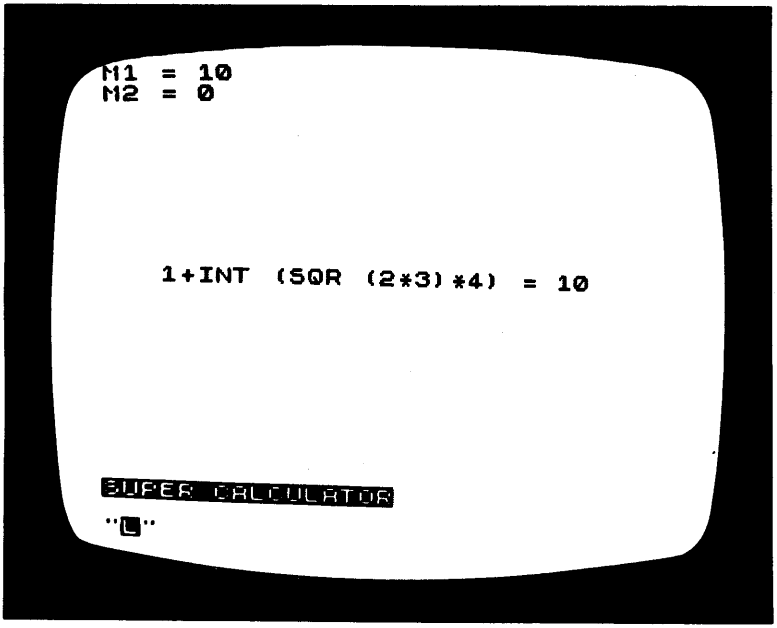
Figure 4.1: The Super-Calculator Program

machines to do our calculations. So the first thing this program tells you, even before the answer to your problem, is the exact expression that you typed into the computer. You can study this echo at your leisure to make sure it actually represents the calculation you wanted to perform. After the echo, the computer gives you the answer to the problem. In this case, the answer is 10.

In addition, this program sets up two memory variables for you. They are called M1 and M2; after the first calculation their current values are always displayed for you in the upper-left corner of your screen. M1 is the result of the most recent calculation the computer has performed. M2 is the result of the calculation just previous to the most recent one. As you can see on your screen, the current values of M1 and M2 are:

```
M1 = 10
M2 = 0
```

The value of M1 is the result of the calculation that is currently



```
M1  =  10
M2  =  0



      1+INT  (SQR  (2*3) *4)  =  10



SUPER CALCULATOR
""
```

Figure 4.2: An Answer from the Super-Calculator Program

displayed in the center of the screen. Since this was the first calculation you have performed during this run of the program, M2 is still equal to zero. (Both M1 and M2 have values of zero at the beginning of the program.)

Now type a second expression for the program to evaluate:

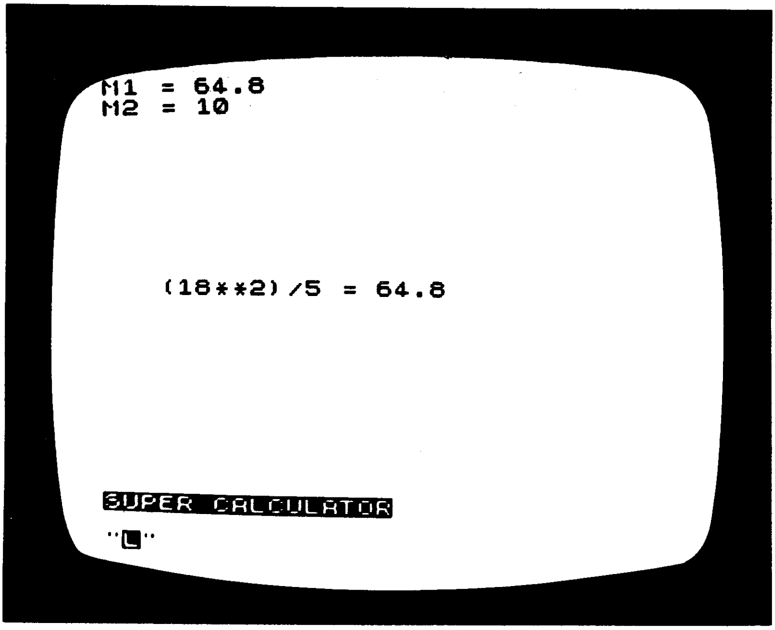
```
(18 ** 2) / 5
```

The answer, as shown in Figure 4.3, is 64.8. Look at what has happened to the two memory variables:

```
M1 = 64.8
M2 = 10
```

The most recent result is shown now as M1. The result of the first calculation, 10, is shown as M2. The purpose of these two memory variables is this: you can use them in expressions that you type into the computer. For example, enter the following expression as your third calculation:

```
M1 / M2
```



```
M1  = 64.8
M2  = 10



      (18**2) /5  = 64.8



SUPER CALCULATOR
```

Figure 4.3: A Second Calculation

This means, "divide the current value of M1 by the current value of M2." The computer will give you the answer:

```
M1 / M2 = 6.48
```

Having access to these two memory variables will often be extremely useful to you when you have a long series of calculations to perform. Their current values will be stored and displayed at each step.

Keep practicing with this program until you understand all of its features. You may find out rather soon that the program run is interrupted, and an error message is displayed, if you type an expression that the computer cannot evaluate to a number. This can happen in a number of ways; any of the following errors will result in termination of the program:

- entering an expression that contains an uneven number of parentheses;
- including an undefined variable in your expression (i.e., any variable other than M1 or M2);
- requesting a calculation that results in division by zero;
- using a function that does not result in a number.

If you make one of these mistakes, you can, of course, get back into the program by simply entering the RUN command. The disadvantage of doing this, however, is that you will lose the current values of the two memory variables, M1 and M2. (The command RUN, you'll recall, clears all current variables before beginning the program.) A better way of restarting the program, then, is to enter the immediate command:

```
GOTO 100
```

When you do this, the unchanged current values of M1 and M2 will appear on the screen, and the program will be ready for your next calculation. We will see exactly why this GOTO command works when we examine the program's inner structure. In general, you can always restart any program by using a GOTO statement instead of RUN. The result is that the current values of any variables in the computer's memory will be maintained.

Now let's take a look at the program itself. You can display the program on your screen by entering the STOP command to end the program run, and then pressing the ENTER key again. Or you can refer back to Figure 4.1, where the program listing appears.

INSIDE THE SUPER-CALCULATOR PROGRAM

The first three lines of the program are straightforward. Line 10 *initializes* the memory variable M1 to zero; line 20 displays the title of the program on the screen; and line 30 reads the input from the keyboard. Notice that the input is stored in the string variable C$:

```
30 INPUT C$
```

The expression is *not* yet evaluated as a calculation, as it would be if the variable were numeric. As far as the computer knows at this point, C$ contains an ordinary string of characters.

Line 40 puts the computer into the fast mode:

```
40 FAST
```

We've already seen the FAST command in action a couple of times in earlier programs in this book, but we've never discussed exactly what it does. In the fast mode the computer stops sending information to the TV screen during the time that it is performing calculations. As a result, calculations are completed much faster. When you are running a program, you can tell it is in the fast mode if the screen goes blank at certain points in the program. In the super-calculator program, you'll recall, the screen goes blank just after you enter the expression you want to calculate. The computer is devoting its resources to calculating, and as a result the job is done quickly.

As an experiment, you might try deleting line 40 and then running the program to see what difference it makes. You'll find that calculations take noticeably longer to perform. (Make sure you put line 40 back into the program before you continue.)

Line 50 clears the screen with the CLS command. Line 60 is the real workhorse statement of this program:

```
60 LET RESULT = VAL C$
```

As we saw in Chapter 3, the VAL function finds the numeric equivalent of a string. In order for VAL to work at all, of course, the string has to be made up of characters that can be converted into numbers; this is why the expressions you type into this program must all be valid numeric expressions. When you make a mistake by typing some invalid expression, the error message you'll get is:

```
C/60
```

The 60 means that the program terminated at line 60. As you can read in Appendix B, the C error code means that you gave an invalid string expression to the VAL function.

If all goes well, then, the LET statement in line 60 stores the calculated result of your numeric expression in the numeric variable RESULT. Lines 70 and 80 reset the values of the memory variables, M1 and M2. First, M2 receives the result of the previous calculation, which is currently stored in M1:

```
70 LET M2 = M1
```

Then M1 receives the result of the latest calculation:

```
80 LET M1 = RESULT
```

The next three lines prepare the screen. Line 90 displays the original expression (still stored in the string variable C$) and the answer (stored in M1):

```
90 PRINT AT 10,3; C$; " = "; M1
```

Lines 100 and 110 display the memory values in the upper-left corner of the screen:

```
100 PRINT AT 0,0; "M1 = "; M1
110 PRINT AT 1,0; "M2 = "; M2
```

This is why you can use the command:

```
GOTO 100
```

to restart the program in the event of an error termination.

Finally, line 120 returns the computer to the SLOW mode and line 130 sends control of the program back up to line 20, forming the endless loop.

Before moving on to the final topic of this chapter, run the super calculator program again for one last experiment. Type the expression:

```
5 ** 20
```

at the keyboard and press the ENTER key. You probably guessed in advance that the expression "5 to the 20th power" would result in a very large number, but the display on the TV screen may be a surprise to you. Here's what it looks like:

```
5 ** 20 = 9.5367432E + 13
```

The answer is expressed in what is called *scientific notation.* The computer automatically switches into this notation for very large numbers like this one. The notation is not as hard to read as it may seem at first if you're not familiar with it. The letter E, toward the end of the answer, stands for exponent, and the expression E + 13 thus means 10 to the 13th power, or:

10,000,000,000,000.

Thus, the full expression given in scientific notation means:

$9.5367432 \times 10{,}000{,}000{,}000{,}000$

or:

95,367,432,000,000

When you consider that the computer can produce numbers that are much larger than this one, you may begin to see the rationale for using scientific notation. It is simply a more economical way of displaying large numbers on the screen.

If you are interested in pursuing this subject further, you can use the super-calculator program to answer a number of questions that might come to mind. With a little experimentation, you can discover many things about how your computer handles numbers:

- As numbers get larger and larger, when does the computer start displaying them in scientific notation?
- What is the largest number your computer can handle?
- How accurately does the computer display very large numbers? (Try typing 5 ** 15. You know that any power of 5 should end in the digit 5. Does the computer's answer end in 5?)

CREATING BAR GRAPHS

The next program we'll look at is designed to produce bar graphs on your TV screen. A bar graph provides a convenient, and often dramatic, means of comparing a series of numbers. Each number is represented by a bar; the larger the number, the longer the bar. When you look at a bar graph, you can see at a glance the relative importance of each category represented on the graph. For this reason, you may want to use the bar graph program to present important information whenever you are more interested in *relative* values than in actual data.

Since the bar graph deals with lists of data, it is time for you to learn about the BASIC data structure that is designed to handle such lists. This structure is called the *array*. We'll look at arrays, and how to define them, before moving on to our program.

Arrays

You know by now that a simple variable can hold exactly one value at a time. If you assign a new value to a variable—using LET or INPUT—the old value will be lost forever. The simple variable is a convenient means of storing any number of data items that are not related to each other in any special way except that they are all needed in the program.

Often, however, when you use your computer to process data, you'll have lists or tables of related data that you'd like to store in the computer's memory. Dozens—or even hundreds—of data items might be involved, so it would be very inconvenient to have to define a separate variable for each item.

BASIC provides special data structures, called *arrays,* that are perfect for storing large quantities of data. Arrays are said to have *dimensions*. A one-dimensional array is used for storing a *list* of data items. A two-dimensional array can store a *table* of data items, arranged in rows and columns. On your T/S 1000 you'll probably seldom need to use arrays that have more than two dimensions, although your BASIC does allow multi-dimensional arrays. The bar graph program uses one-dimensional arrays, so we will limit our discussion to storing *lists* of data.

Suppose you have a list of 15 numbers that you want to enter into your computer's memory. You want to design a program that will perform three tasks:

1. read the numbers from the keyboard during the program run;
2. perform some calculations on the numbers;
3. display the numbers again on the screen along with the results of the calculations.

You can begin by defining an array that will hold all 15 of the numbers at once. Let's say you decide to call the array N (for *numbers*). Here's how you would define the array, in the first line of your program:

```
10 DIM N(15)
```

The keyword DIM, located above the [D] key, stands for dimension. A DIM statement specifies the four essential characteristics of an array:

1. the name of the array
2. the type of data it will hold (numeric or string)
3. the number of dimensions it has
4. the number of data items allowed in each dimension. (This last characteristic is referred to as the *length.*)

Here are the characteristics of the array defined in line 10, above:

1. its name is N;
2. it is a numeric array (since its name does not end in a $ character);
3. it has one dimension (since there is only one number specified inside the parentheses);
4. the length of the array is 15.

When the computer comes to this line in your program, it automatically sets aside room in its memory for 15 numbers under the array name N.

With the array defined, you need some way to identify each of its 15 elements. Once you've assigned numbers to the array, you might want to access the 7th number in the list, for example; if so, you'll need a convenient way to tell the computer exactly which element you're interested in.

The system for identifying the elements of an array is very simple. You write the array name, then, in parentheses, the number of the element you want to identify. For example, the 7th element of the array N is named:

```
N(7)
```

Likewise, the first 5 elements are:

```
N(1)
N(2)
N(3)
N(4)
N(5)
```

The number in parentheses is called the *index* into the array. You can write statements like:

```
LET N(7) = 162
INPUT N(5)
PRINT N(2)
```

The first of these statements stores the number 162 as the 7th element of the array N. The second statement will read a number from the keyboard, which will be stored in the 5th element of N. The third statement displays the current value of the 2nd element of N on the screen.

You can also write a variable name between the parentheses after the array name; for example:

```
N(I)
```

Then the choice of elements depends on the value of the index I. If I equals 5, N(I) refers to the 5th element of N. Knowing this, you'll be able to use FOR loops in which the control variable of the loop is also used as the index into the array. For example:

```
10 DIM N(15)
20 FOR I = 1 TO 15
30 INPUT N(I)
40 NEXT I
```

In this program, the control variable, I, is also the index into the array N. With these simple lines, you can instruct the computer to read fifteen numbers from the keyboard and to store them all under the array name N. You can begin to see what a powerful set of tools we have in arrays and FOR loops.

To see this power in action, type lines 10 to 40, above, into your computer. Add the following three lines, which display all fifteen values of N on the screen:

```
50 FOR I = 1 TO 15
60 PRINT N(I)
70 NEXT I
```

Run the program. Enter any fifteen numbers into the computer; when you're done, they'll all be displayed at once on the screen. The important thing to remember is that all fifteen numbers are stored in the computer's memory, and can be accessed and used for whatever purpose suits you. Any operation that must be performed on a series of items can be incorporated into a FOR/NEXT loop.

Your computer also allows you to define string arrays, but the method is slightly more complicated than for numerical arrays. Again, we will look at a one-dimensional array. In the DIM statement, after you have defined the dimension and its length, you must add another number within the parentheses. This number represents the length of each element of the string array, the number of characters it contains. The elements of a string array are all of the same predefined length. Let's look at an example.

Suppose you have a list of fifteen words you want to store in the computer's memory for use in a program. The longest word contains eight characters. You can define a string array for this list as follows:

```
10 DIM W$(15,8)
```

The name of the array is W$; it will hold 15 strings, each 8 characters long.

You can experiment with this array by typing in the following short program:

```
10 DIM W$(15,8)
20 FOR I = 1 TO 15
30 INPUT W$(I)
40 PRINT W$(I);
50 NEXT I
```

Notice, first of all, that you can refer to elements in this string array in the same way you used the numerical array. The 7th word in the array W$ is called:

```
W$(7)
```

and the Ith word is:

```
W$(I)
```

Now run the program, and watch what it does. First enter a word that is exactly eight characters long:

```
COMPUTER
```

The word will be stored in W$(1) and, thanks to line 40, it will also be displayed at the top of the screen. Now enter two words, to be stored in W$(2) and W$(3), respectively; choose words that are less than eight characters long:

```
BOOK
PEN
```

The first word will appear on the screen, directly after the word COMPUTER. (Notice that line 40 ends in a semicolon, which prevents a line feed.) The word PEN, however appears four spaces after the word BOOK. You'll see the following line on the screen:

```
COMPUTERBOOK    PEN
```

Why are there four spaces separating BOOK and PEN? Because you defined the elements of the array W$ as having eight characters. Since BOOK contains only four characters, the computer fills W$(2) with spaces to give it the correct length of eight.

Now try entering a word that is *longer* than eight characters:

```
INTERESTING
```

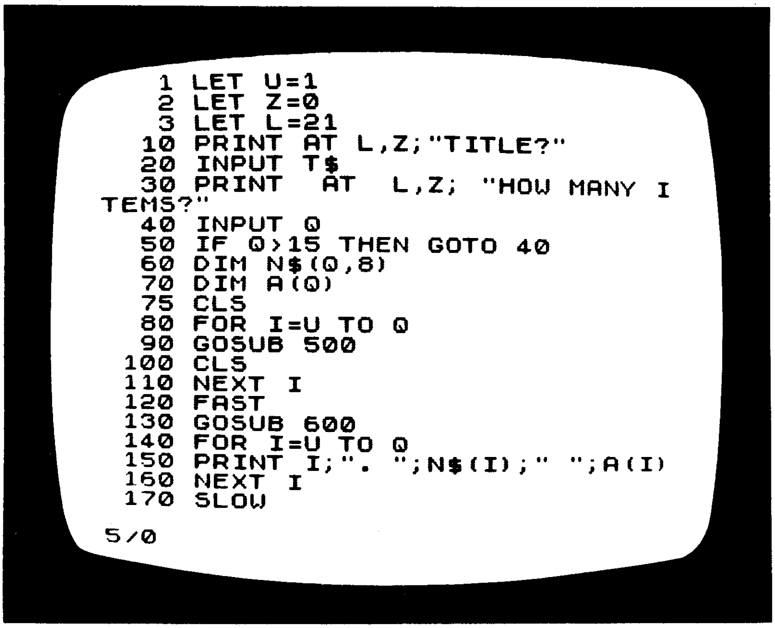
You'll see only the first eight characters, INTEREST, appear on the screen. The whole word cannot fit into an element of W$, so the last three characters are simply chopped off.

Now that you've had a brief introduction to both numerical and string arrays, you're ready to look at the bar graph program. This program uses two different arrays—defined in two DIM statements—to store all the information for creating a bar graph.

THE BAR GRAPH PROGRAM

The listing of the bar graph program appears in Figures 4.4 to 4.7. When you're typing a long program like this into your computer, it's a good idea to save it on cassette in stages. As soon as you've typed about a fourth of the program, save it. Then continue typing until you've finished about half the program. When you perform the second save operation, you can record over the same part of the cassette tape on which you saved the first part of the program, since you don't need the first version any more. Continue saving in steps like this until you've typed and saved the entire program. This process guards against accidental loss of your work. The most you can lose at any point as you are typing is anything you've entered since the previous save operation.

When you've entered the entire program, check through it carefully to make sure you haven't made any mistakes. Then run the program.

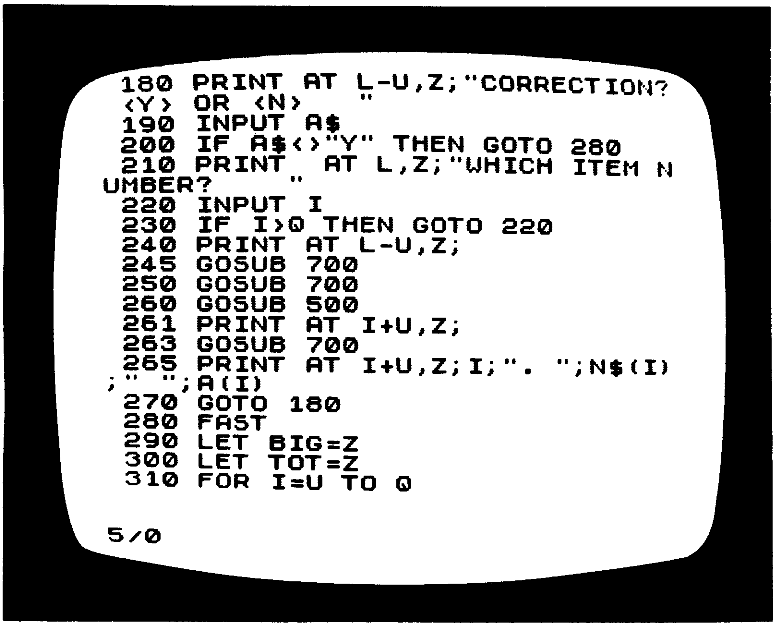


```
   1 LET U=1
   2 LET Z=0
   3 LET L=21
  10 PRINT AT L,Z;"TITLE?"
  20 INPUT T$
  30 PRINT  AT  L,Z; "HOW MANY I
TEMS?"
  40 INPUT Q
  50 IF Q>15 THEN GOTO 40
  60 DIM N$(Q,8)
  70 DIM A(Q)
  75 CLS
  80 FOR I=U TO Q
  90 GOSUB 500
 100 CLS
 110 NEXT I
 120 FAST
 130 GOSUB 600
 140 FOR I=U TO Q
 150 PRINT I;".  ";N$(I);"  ";A(I)
 160 NEXT I
 170 SLOW

5/0
```

Figure 4.4: The Bar Graph Program

This program produces a bar graph for any list of numbers up to fifteen items long. Figure 4.8 shows an example of the end result of the program. This particular bar graph compares grocery expenses, item by item, for the month of December. If these were real data from your own household, you would have an easy-to-read graphic representation of your expenses for the month. (If you're scratching your head and wondering what you would *do* with a bar graph of grocery expenses, stop worrying. This program can be used for *any* set of data, involving any subject. At the beginning of the program you supply the title of the bar graph. You also supply the name of each category along with the numerical data.)

This program guides you through three distinct phases of activity in order to produce the bar graph: First, you must input all the data—including title, item names, and numerical data. The program supplies clear prompts to tell you exactly what kind of information to enter at each point of the input phase. The second phase is for the correction of any input errors. This phase begins with a complete "echo," in table form, of all the data you have entered



```
 180 PRINT AT L-U,Z;"CORRECTION?
 <Y> OR <N>       "
 190 INPUT A$
 200 IF A$<>"Y" THEN GOTO 280
 210 PRINT   AT L,Z;"WHICH ITEM N
UMBER?     "
 220 INPUT I
 230 IF I>Q THEN GOTO 220
 240 PRINT AT L-U,Z;
 245 GOSUB 700
 250 GOSUB 700
 260 GOSUB 500
 261 PRINT AT I+U,Z;
 263 GOSUB 700
 265 PRINT AT I+U,Z;I;".  ";N$(I)
;"  ";A(I)
 270 GOTO 180
 280 FAST
 290 LET BIG=Z
 300 LET TOT=Z
 310 FOR I=U TO Q

5/0
```

Figure 4.5: The Bar Graph Program (cont.)

into the computer. You can study this information at your leisure, and you'll have the opportunity to correct any errors that you find in the input data. Finally, in the third phase, you'll see the bar graph produced from your data. Let's examine each of these phases; we'll enter the grocery expense data in this sample run through the program.

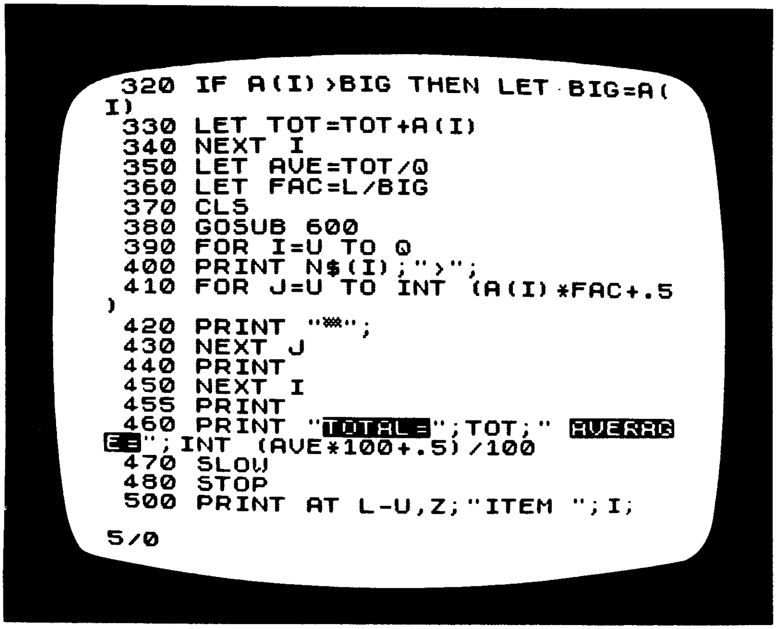
At the beginning of the program, as you've already seen on your own screen, the program prompts you to enter the title of the bar graph:

```
TITLE?
```

You may enter any title up to 32 characters long. Enter the title you see at the top of the bar graph in Figure 4.8. Following this prompt you will see a second question on the screen:

```
HOW MANY ITEMS?
```

You must enter the number of items there will be in your bar graph (up to 15 items). In the grocery expense graph there are 15 items,



```
 320 IF A(I)>BIG THEN LET BIG=A(
I)
 330 LET TOT=TOT+A(I)
 340 NEXT I
 350 LET AVE=TOT/Q
 360 LET FAC=L/BIG
 370 CLS
 380 GOSUB 600
 390 FOR I=U TO Q
 400 PRINT N$(I);">";
 410 FOR J=U TO INT (A(I)*FAC+.5
)
 420 PRINT "▒";
 430 NEXT J
 440 PRINT
 450 NEXT I
 455 PRINT
 460 PRINT "TOTAL =";TOT;" AVERAG
E=";INT (AVE*100+.5)/100
 470 SLOW
 480 STOP
 500 PRINT AT L-U,Z;"ITEM ";I;

5/0
```

Figure 4.6: The Bar Graph Program (cont.)

so you should type the number 15 and then press ENTER.

Next, the program will ask you to enter the name, and the numerical amount for each of the 15 items of your graph. The prompt for the first item's name will look like this:

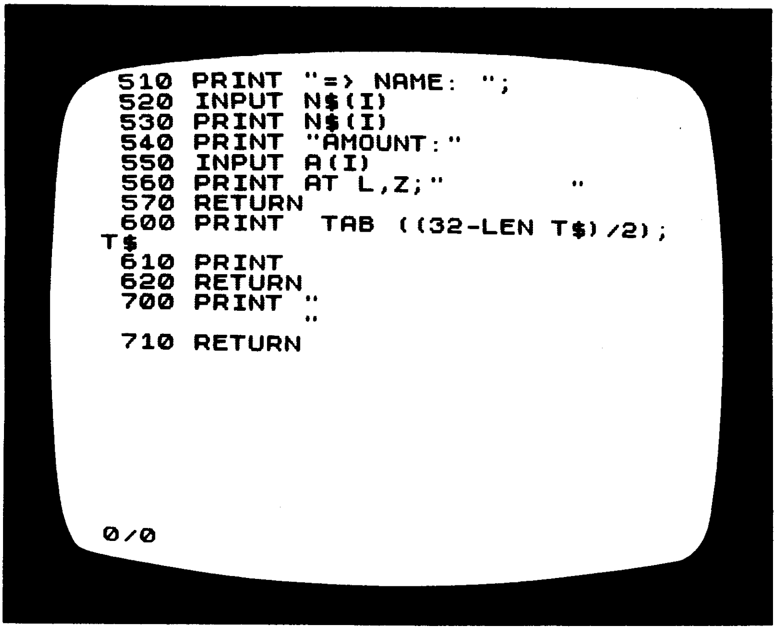
```
ITEM 1 = > NAME:
```

You should enter the word MEAT, the first grocery item. As soon as you've done so, the prompt will change to:

```
ITEM 1 = > NAME: MEAT
AMOUNT:
```

Now you must enter the amount spent on meat for the month. Enter the number 27.53.

The program will continue to ask you for the name and amount of each item until you have entered all 15 items. Figure 4.9 shows the items you should enter after MEAT. (Notice that the word "vegetables" has been abbreviated to 8 characters, the maximum string length for the data items in this program.)



```
 510 PRINT "=> NAME:  ";
 520 INPUT N$(I)
 530 PRINT N$(I)
 540 PRINT "AMOUNT:"
 550 INPUT A(I)
 560 PRINT AT L,Z;"          "
 570 RETURN
 600 PRINT  TAB ((32-LEN T$)/2);
T$
 610 PRINT
 620 RETURN
 700 PRINT "
"
 710 RETURN




0/0
```

Figure 4.7: The Bar Graph Program (cont.)

When you have entered the last item, the screen will go blank for a moment. Then all the information shown in Figure 4.9 will appear on the screen. This is the "echo" of all the input data that you supplied to the program. At the bottom of the screen, just after the table of data, appears the question:

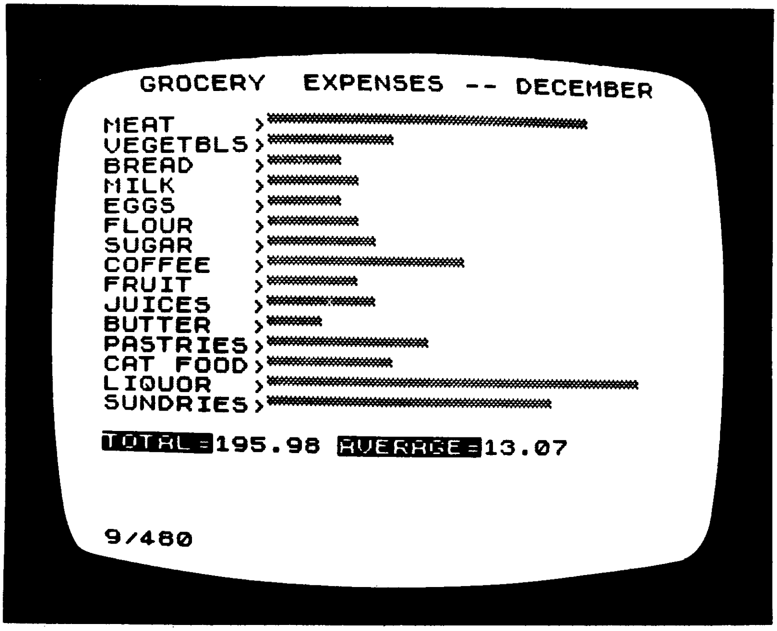
```
CORRECTION? <Y> OR <N>
```

This is your chance to examine the data for any typing errors that you might have made during the input phase. If you find an error, you need only type Y (for "yes") to indicate that you have a correction to make.

In response, the program will ask:

```
WHICH ITEM NUMBER?
```

At this point you must type the number of the item that you wish to change. To see how the correction process works, let's change the amount shown for item #9, FRUIT. Enter the number 9. The program will then prompt you to enter the name and number of



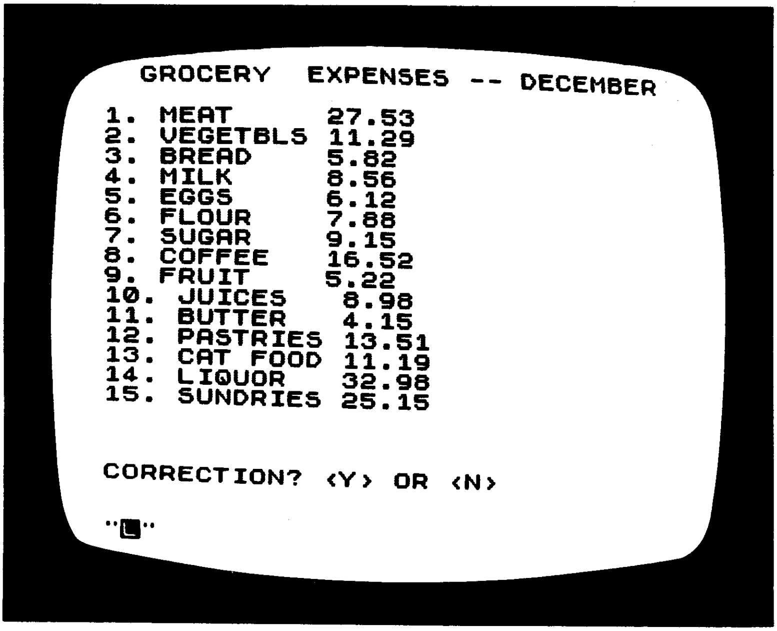
```
      GROCERY   EXPENSES -- DECEMBER
MEAT      >
VEGETBLS  >
BREAD     >
MILK      >
EGGS      >
FLOUR     >
SUGAR     >
COFFEE    >
FRUIT     >
JUICES    >
BUTTER    >
PASTRIES  >
CAT FOOD  >
LIQUOR    >
SUNDRIES  >

TOTAL =195.98 AVERAGE =13.07


9/480
```

Figure 4.8: Sample Run of the Bar Graph Program: Grocery Expenses

the item. (You must enter both pieces of information, even if you only want to change one of them.) Enter FRUIT for the name, and the new value 7.15 for the amount. The correction will be made on the table display, so you can see the results of your correction immediately. You can make as many corrections as you want; the correction phase continues until you enter the letter N (for "no") in response to the correction prompt.

As soon as you leave the correction phase of the program, the screen will go blank for a few moments, and then the bar graph will appear on the screen. Take another look at Figure 4.8, the bar graph for the grocery expense data. Notice that, in addition to the graph, the program gives you the total amount spent for the month, and the average expense per item.

Figure 4.10 shows a second bar graph example. This time the title is "Office Supplies—Expenses, 1982" and the bars represent monthly expenses for office supplies, from January to December. The total expense for the year is shown as $7,439.86, and the average monthly expense as $619.99.



```
   GROCERY  EXPENSES -- DECEMBER
 1. MEAT        27.53
 2. VEGETBLS    11.29
 3. BREAD       5.82
 4. MILK        8.56
 5. EGGS        6.12
 6. FLOUR       7.88
 7. SUGAR       9.15
 8. COFFEE      16.52
 9. FRUIT       5.22
 10. JUICES      8.98
 11. BUTTER      4.15
 12. PASTRIES    13.51
 13. CAT FOOD    11.19
 14. LIQUOR      32.98
 15. SUNDRIES    25.15


 CORRECTION? <Y> OR <N>

 "■"
```

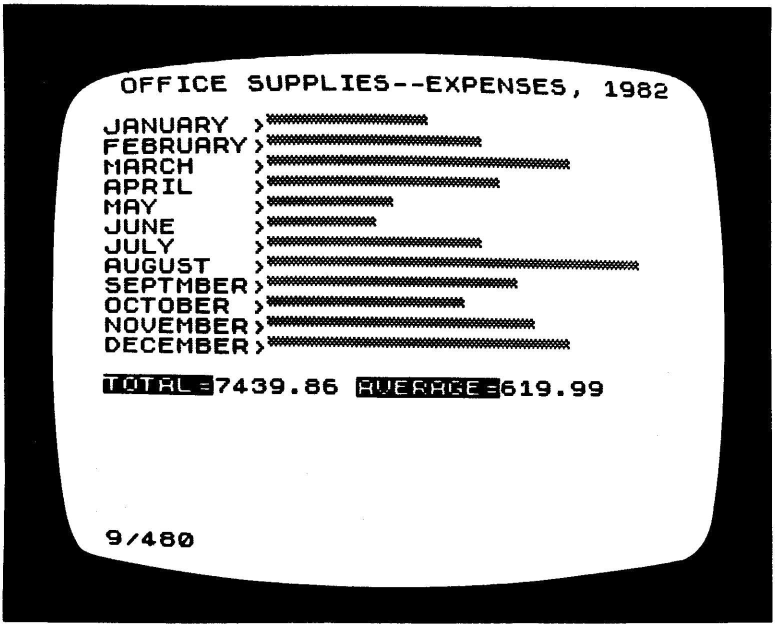
Figure 4.9: The Grocery Data

In the next section of this chapter we'll examine the structure of this program. Before you read on, you might want to run the program again, entering some real data of your own (from your home, business or school life) and produce a bar graph that has meaning to you.

INSIDE THE BAR GRAPH PROGRAM

Once again, as you read through this description, you can either refer back to Figures 4.4 through 4.7, where the listing is presented; or you can list the program, section by section, on your own screen.

This program comes closer than others we've discussed to filling up the memory available in your computer (2K only; not the extra memory module). For this reason, three variables are defined at the beginning of the program that act as "constants" throughout the program; as we saw in Chapter 3, this is a method of saving



```
 OFFICE SUPPLIES--EXPENSES, 1982
JANUARY  >
FEBRUARY >
MARCH    >
APRIL    >
MAY      >
JUNE     >
JULY     >
AUGUST   >
SEPTMBER >
OCTOBER  >
NOVEMBER >
DECEMBER >

TOTAL =7439.86 AVERAGE =619.99




9/480
```

Figure 4.10: Second Bar Graph Example: Office Supplies

memory space by replacing literal numeric values with single-letter variable names:

```
1 LET U = 1
2 LET Z = 0
3 LET L = 21
```

The variable L is used as an address for placing prompts on the screen.

As another way of conserving memory, no REM lines have been placed in the program. This is unfortunate in such a long program; but it is a choice you will sometimes have to make. To compensate for the lack of comments in the listing, here is a section-by-section outline of the program:

- *Initialization section* (lines 1 to 70).

This section defines the three "constant" variables; reads the title and the number of items from the keyboard; and dimensions two arrays, N$ and A, to store the bar graph data.

- *Input section* (lines 75 to 110).

This section controls the input of all the bar graph data. It calls the subroutine at line 500 once for each item. The subroutine at line 500 in turn displays the prompts on the screen and reads input from the keyboard.

- *Echo section* (lines 120 to 170).

This section "echoes" all the input data on the screen in one complete table. It calls the subroutine at line 600 to display the title at the top of the screen.

- *Correction section* (lines 180 to 270).

This section controls the correction dialogue. Whenever you correct an item, this section stores the new data in the appropriate elements of N$ and A, and displays the new data on the screen. The correction section calls two subroutines: the subroutine at 500, which supplies the input prompts and reads data from the keyboard; and the subroutine at line 700, which clears a single line of the screen so that new information can be displayed there.

- *Intermediate calculations section* (lines 280 to 360).

This section calculates four values from the numerical data stored in the array A. BIG is the largest value in the array. TOT is the sum of all the values in the array. AVE is the average of the values in the array. The value FAC is a conversion factor that will be used to make each bar of the graph proportional in length to the number it represents.

- *The bar graph section* (lines 370 to 480).

This section begins with a call to the subroutine at line 600, which displays the title of the bar graph. Then it draws each bar, calculating the appropriate length using the value of the conversion factor FAC. Finally, this section displays the total and average values of the data.

- *The subroutines*

— at line 500: the input subroutine;
— at line 600: the title subroutine;
— at line 700: the clear-line subroutine.

Using the outline above as a guide, you should be able to read through the entire program, and understand the meaning and purpose of most of the lines. A few of the lines may still need some explanation, however. The following notes will clarify several of the program's more difficult lines:

Lines 60 and 70 contain the DIM statements. Line 60 defines the string array, N$, to store the names of the bar graph items, and line 70 defines the numerical array, A, for the amount of each item:

```
60 DIM N$(Q,8)
70 DIM A(Q)
```

The *length* of each of these arrays is specified by the variable Q. During the program run, you input a value for this variable in answer to the question, "HOW MANY ITEMS?":

```
30 PRINT AT L, Z; "HOW MANY ITEMS?"
40 INPUT Q
```

If, for example, you input the number 12 for the quantity of items in the bar graph, then the length of each array will be set at 12, as though lines 60 and 70 were written as follows:

```
 60 DIM N$(12,8)
 70 DIM A(12)
```

Through use of the variable Q, then, you know that the memory space reserved for the two arrays will be exactly the amount needed, no more and no less.

These two arrays are assigned values in the FOR loop located at lines 80 to 110:

```
 80 FOR I = U TO Q
 90 GOSUB 500
100 CLS
110 NEXT I
```

The FOR line varies the control variable, I, from 1 to the value of Q. The subroutine at line 500 uses this control variable to specify the correct item number in the message it displays on the screen:

```
500 PRINT AT L - U,Z; "ITEM ";I;
```

and also to store each item of input data in the correct element of the appropriate array:

```
520 INPUT N$(I)
 .
 .
 .
550 INPUT A(I)
```

These input lines, and the prompt lines that precede them, are isolated in their own subroutine because they are needed at two different times in the program: first in the initial input section; then in the correction section.

Another FOR loop, in lines 140 to 160, displays all the information from both arrays on the screen:

```
140 FOR I = U TO Q
150 PRINT I; ". "; N$(I); " "; A(I)
160 NEXT I
```

Line 150 uses the control variable I three times: first to display the item number, then to access the correct element of N$, for the name of the item; and finally to display the amount of the item, from the array A.

Lines 310 to 340 determine the largest value in the array A

(BIG), and the sum of all the values (TOT). The two variables used for these two calculations are first initialized to zero:

```
290 LET BIG = Z
300 LET TOT = Z
```

Then another FOR loop moves through the entire array from 1 to Q:

```
310 FOR I = U TO Q
320 IF A(I) > BIG THEN LET BIG = A(I)
330 LET TOT = TOT + A(I)
340 NEXT I
```

Line 320 compares each value of the array to the current value of BIG. Each time it finds a value larger than BIG, that value becomes the new BIG. Line 330 accumulates a running total of all the values.

The average value is the total divided by the number of values, Q:

```
350 LET AVE = TOT / Q
```

The proportional conversion factor is based on the largest value in the array, BIG:

```
360 LET FAC = L / BIG
```

The variable L, you'll recall, is a "constant" value, equal to 21. Thus, the longest bar in the graph will be 21 characters long. As we'll soon see, multiplying each value in the array A by the conversion factor FAC will result in a number between zero and 21; this number, in turn, will determine the length of the corresponding bar in the graph.

The bar graph itself is displayed on the screen by lines 390 to 450. These lines are worth careful examination. They show you an example of a FOR loop *within* another FOR loop. We sometimes call this a *nested loop*; there are two essential rules that you must be aware of for the design of nested loops:

1. The inner loop must use a different control variable than the outer loop.
2. The FOR and NEXT statements of the inner loop must *both* be located inside the outer loop.

Keep these rules in mind as you examine the lines that create the bar graph.

The outer loop, which begins at line 390, controls the item number, incrementing the control variable I from 1 to Q:

```
390 FOR I = U TO Q
```

First, the name of the item is displayed:

```
400 PRINT N$(I);"> ";
```

Then the inner loop creates a bar of correct proportional length to the amount, A(I):

```
410 FOR J = U TO INT (A(I) * FAC + .5)
```

The expression A(I) * FAC will result, as we have seen, in a number between 0 and 21. By adding .5 and taking the integral value of the result, we can round this number off to the nearest integer:

```
INT (A(I) * FAC + .5)
```

In this way the inner loop increments its control variable, J, from 1 to the appropriate length of the bar that is to represent the value A(I). Each time J is incremented, line 42 increases the length of the bar by one character. (The graphics character that is the "building block" of the bars appears on the [S] key of your keyboard.)

To summarize, you can see that for each repetition of the outer loop, the inner loop goes to work creating one bar of the bar graph. These are probably the most important lines for you to understand in this program. If you're having trouble seeing how they work, perhaps the following analogy will help you grasp the concept of a nested loop.

Think of the action of the three hands on a clock. If you define one time around the clock as a "loop," you can see three levels of loops on the clock. The innermost loop is the second hand, which repeats its action 60 times for every complete loop of the minute hand. The minute hand, in turn, goes around the clock 12 times for every loop of the hour hand. Thus, there are three levels of repetition; the innermost loop sees the most activity, and the outermost loop controls all the activity of the other two loops. The action of the nested loops in our program is similar: The outer loop at line 390 controls the activity of the inner loop at line 410.

SUMMARY

Your computer offers you five arithmetic operations (+ , − , *, /, **) and a useful assortment of mathematical functions. You can combine these operations and functions in complex expressions in order to perform calculations. In writing such expressions, you must always be aware of the computer's established order of operations. To override this order—or simply to clarify an expression and make it more readable—you can include parentheses in arithmetic expressions.

The computer's fast mode can be useful when you want the computer to concentrate on performing rapid calculations, as we saw in the super-calculator program.

The *array* is an important data structure in BASIC that allows you to store many values under one variable name. The array is an ideal tool to use within FOR loops; the control variable of a FOR loop can be used as an *index* into the array.

The bar graph program presented in this chapter shows a good example of the power of nested FOR loops, as well as several examples of the efficient use of arrays within FOR loops.

Chapter 5

Words, Words, Words: Strings and String Functions on Your Computer

INTRODUCTION

In this final chapter we'll take a closer look at strings and the functions and operations you can perform on them. As you know, a string is a data item that consists of one or more characters. You've already had considerable experience with strings and string variables in the programs presented in earlier chapters; but there are still several important functions that you should learn about.

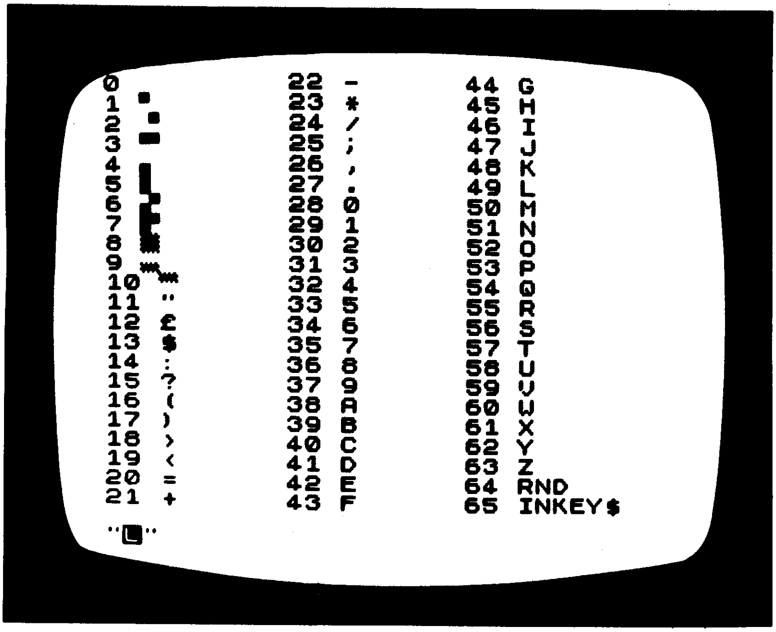
We'll begin this chapter with a look at the character code used by the T/S 1000. You'll see how to use the functions CODE and CHR$ to convert back and forth between the code numbers and the characters they represent. You'll examine and run a couple of short programs along the way; you'll also learn the meaning of concatenation, and experiment with the useful LEN function.

Next, in preparation for the last program in this book, you'll find out what happens when you "slice" a string and why you might want to do so. The last program, called the decision-maker program, is an amusing item that turns the computer into your own private counselor and adviser. Any way you slice it, this program wouldn't think of stringing you along.

THE T/S 1000 CHARACTER CODE

The computer uses its own private code to store—and distinguish among—the characters on your keyboard. In some programming situations, it can be useful for you to know about this code.

In the code, each character is assigned a number between 0 and 255. In this context, the computer considers every item on your keyboard a *character*—even the keywords, functions, and other words that appear on the keyboard. Figures 5.1 to 5.3 show you the code. There is certainly no reason for you to memorize any part of this code; but you should take a moment to become familiar with it and take note of the general location of the several categories of characters. For example, you can see that the graphics characters are located in two separate parts of the code (1 to 10, and 128 to 138); the digits and letters are located near the beginning of the code (28 to 63); and the reverse-video characters appear in about the middle of the code (139 to 191). You can also find all the



```
0             22 -          44 G
1 ▘           23 *          45 H
2 ▝           24 /          46 I
3 ▀           25 ;          47 J
4 ▖           26 ,          48 K
5 ▌           27 .          49 L
6 ▞           28 0          50 M
7 ▛           29 1          51 N
8 ▒           30 2          52 O
9 ▒           31 3          53 P
10▒           32 4          54 Q
11 "          33 5          55 R
12 £          34 6          56 S
13 $          35 7          57 T
14 :          36 8          58 U
15 ?          37 9          59 V
16 (          38 A          60 W
17 )          39 B          61 X
18 >          40 C          62 Y
19 <          41 D          63 Z
20 =          42 E          64 RND
21 +          43 F          65 INKEY$
 "L"
```

Figure 5.1: The Character Code

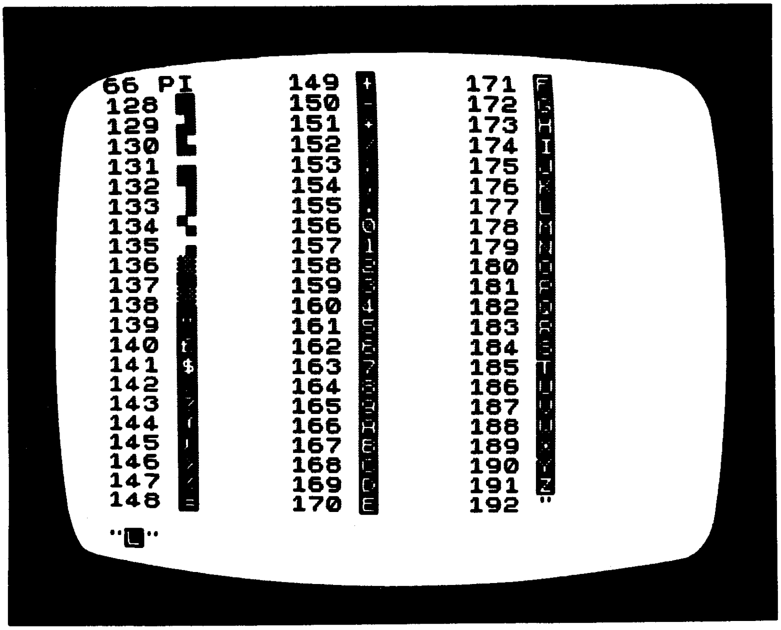
keywords, functions, operators, and other symbols in the code.

You may notice that a large section of the code is missing (from 67 to 127). Most of these missing code numbers are simply unused; a few of them are used for "control keys," such as the direction key and the DELETE key, which do not display characters on the screen.

The CHR$ function supplies the character represented by a code number. The number must, of course, be between 0 and 255; CHR$ returns the corresponding character, keyword, function, or symbol.

The following six-line program reads a code number from the keyboard and then displays the character on the screen for you:

```
10 PRINT AT 21,0; "WHAT CODE NUMBER?"
20 INPUT C
30 IF C > 255 THEN GOTO 20
40 CLS
```



```
66 PI        149 +       171 F
128          150 -       172 G
129          151 *       173 H
130          152 /       174 I
131          153 ;       175 J
132          154 ,       176 K
133          155 .       177 L
134          156 0       178 M
135          157 1       179 N
136          158 2       180 O
137          159 3       181 P
138          160 4       182 Q
139 "        161 5       183 R
140 £        162 6       184 S
141 $        163 7       185 T
142 :        164 8       186 U
143 ?        165 9       187 V
144 (        166 A       188 W
145 )        167 B       189 X
146 >        168 C       190 Y
147 <        169 D       191 Z
148 =        170 E       192 "
"L"
```

Figure 5.2: The Character Code (cont.)

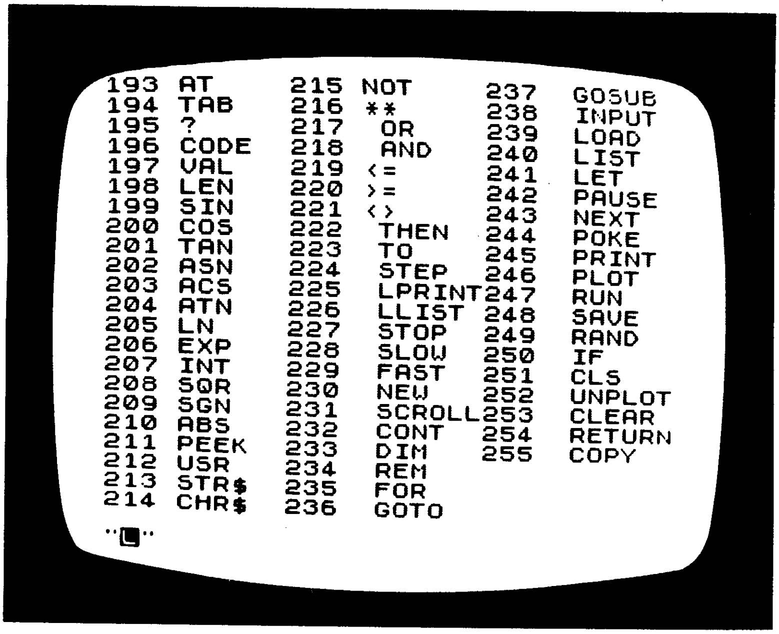
```
50 PRINT AT 8,8; "CODE ";C; " = ";CHR$ C
60 GOTO 10
```

Line 30 tests the number you type to make sure it is a valid code number. Line 50 uses the CHR$ function to display the character.

The CODE function, conversely, supplies the code number of a character. You might recall CODE and CHR$ from Chapter 2; the very first program you typed into the computer used both functions to change a string of letters into its reverse-video equivalent.

The program shown in Figure 5.4 converts strings into reverse video. The program is merely an exercise; it doesn't really do anything useful. But you can learn several interesting techniques for dealing with strings if you take the time to study the program carefully.

Some sample output from this program appears in Figure 5.5. As you can see, the program prompts you repeatedly to enter a string of characters into the computer. After each string input, the program displays the string in reverse-video. The process of creating



```
193 AT      215 NOT     237 GOSUB
194 TAB     216 **      238 INPUT
195 ?       217  OR     239 LOAD
196 CODE    218  AND    240 LIST
197 VAL     219 <=      241 LET
198 LEN     220 >=      242 PAUSE
199 SIN     221 <>      243 NEXT
200 COS     222  THEN   244 POKE
201 TAN     223  TO     245 PRINT
202 ASN     224  STEP   246 PLOT
203 ACS     225  LPRINT247 RUN
204 ATN     226  LLIST  248 SAVE
205 LN      227  STOP   249 RAND
206 EXP     228  SLOW   250 IF
207 INT     229  FAST   251 CLS
208 SQR     230  NEW    252 UNPLOT
209 SGN     231  SCROLL253 CLEAR
210 ABS     232  CONT   254 RETURN
211 PEEK    233  DIM    255 COPY
212 USR     234  REM
213 STR$    235  FOR
214 CHR$    236  GOTO
""
```

Figure 5.3: The Character Code (cont.)

a reverse-video string, performed by lines 50 to 80, is what is interesting about this program.

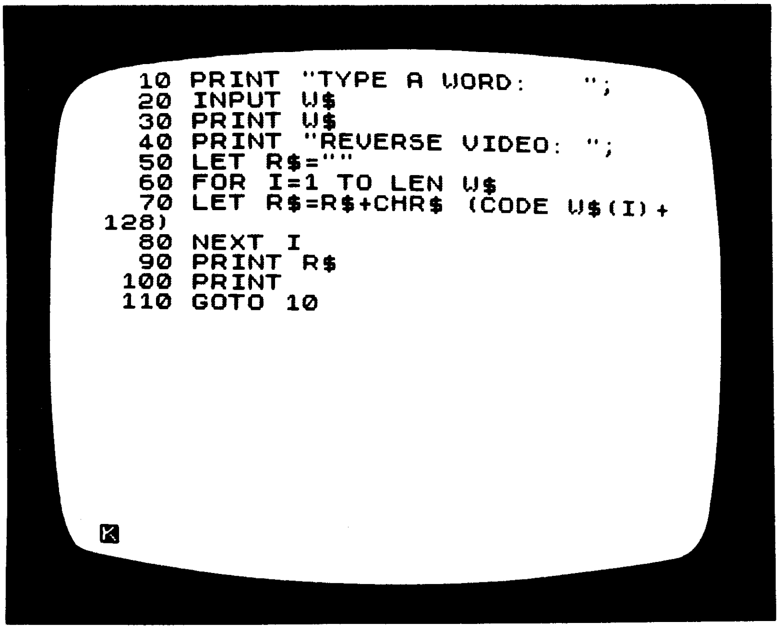
The string variable W$ contains your input word. The variable R$ will hold the reverse-video of the word. In line 50, R$ is initialized as a *null string,* that is, a string that has no characters in it.

```
50 LET R$ = ""
```

Then the FOR loop in lines 60 to 80 finds the reverse-video equivalent of each character of W$, and adds the characters on to the end of R$. The FOR statement (line 60) uses the LEN function to determine the length, in characters, of the input string, W$. As you can see, the control variable, I, is incremented from 1 to LEN W$:

```
60 FOR I = 1 TO LEN W$
```

This is how the loop deals with each character of W$, one character at a time. BASIC will allow you to access *one character* of a string by putting a number representing the position of the character, in



```
  10 PRINT "TYPE A WORD:    ";
  20 INPUT W$
  30 PRINT W$
  40 PRINT "REVERSE VIDEO: ";
  50 LET R$=""
  60 FOR I=1 TO LEN W$
  70 LET R$=R$+CHR$ (CODE W$(I)+
128)
  80 NEXT I
  90 PRINT R$
 100 PRINT
 110 GOTO 10
```

Figure 5.4: The Reverse Video Program

parentheses, after the string variable name. Thus, W$(I) represents the character of W$ located at position I.

For example, if you type a three-character string into the computer, the FOR loop will increment I from 1 to 3. The three characters of the string are W$(1), W$(2), and W$(3).

Now let's look at the expression, in line 70, that converts a character to its reverse-video equivalent:

```
CHR$(CODE W$(I) + 128)
```

Inside the parentheses, the CODE function supplies the code number of the character W$(I). Look once again at the character code in Figures 5.1 to 5.3. If your input string consists of punctuation marks, digits, or letters, the code number for each character of the string will fall somewhere between 11 and 63. If you add 128 to any of these code numbers, you'll see that the resulting code will represent the reverse-video equivalent of the character. Try an example. The character code for the letter D is 41. Adding 128 to 41 gives 169; sure enough, the reverse-video D character has the code 169.

So now you can see exactly what the conversion expression in line 70 does. It begins by adding 128 to the code number of the character in question:

```
(CODE W$(I) + 128)
```

and then uses the CHR$ function to find the character represented by the new code:

```
CHR$ (CODE W$(I) + 128)
```

When the new character is determined, it is added onto the reverse-video string that is accumulating in the variable R$. Combining two strings in this fashion is called *concatenation.* The process is indicated by the plus symbol (+), as in line 70:

```
70 LET R$ = R$ + CHR$(CODE W$(I) + 128)
```

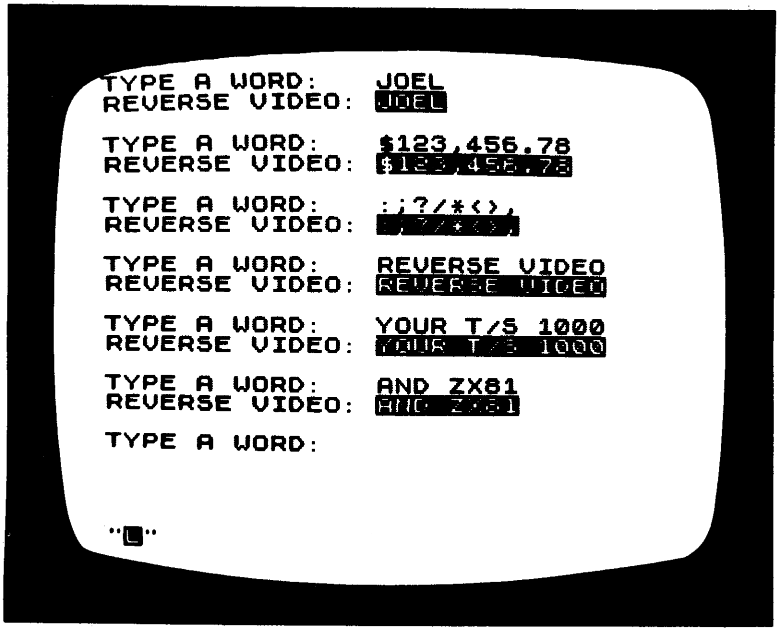
This LET statement says, "Add the new character onto the end of the current string value stored in R$; then store the new string, which is one character longer than before, in R$ again."

The reverse-video conversion is complete when I equals LEN W$; the action of the FOR loop is over, and line 90 displays R$ on the screen:

```
90 PRINT R$
```

To summarize, lines 50 to 80 of this program can teach you several important points about strings and string handling:

- You can initialize a string variable as a *null string,* meaning a string that contains no characters (line 50).
- The function LEN supplies the length, in characters, of a string (line 60).
- You can access a single character of a string by specifying the position of the character, in parentheses (line 70).
- The codes of the reverse-video characters are all numbered higher by 128 than the codes of the regular characters (line 70).



```
TYPE A WORD:      JOEL
REVERSE VIDEO:    JOEL

TYPE A WORD:      $123,456.78
REVERSE VIDEO:    $123,456.78

TYPE A WORD:      :;?/*<>,
REVERSE VIDEO:    :;?/*<>,

TYPE A WORD:      REVERSE VIDEO
REVERSE VIDEO:    REVERSE VIDEO

TYPE A WORD:      YOUR T/S 1000
REVERSE VIDEO:    YOUR T/S 1000

TYPE A WORD:      AND ZX81
REVERSE VIDEO:    AND ZX81

TYPE A WORD:

"L"
```

Figure 5.5: Sample Output from the Reverse Video Program

- You can concatenate two strings (i.e., combine them), using the " + " symbol (line 70).

In the next section you'll see that your computer also provides a way to access several consecutive characters of a string, in a process called *slicing*.

SLICING A STRING

It's often convenient to be able to identify a *substring* of a larger string, for any of several reasons. You might want to examine a string to see if it contains a certain sequence of characters, or you might want to change the values of certain positions inside a string. In the decision-maker program, at the end of this chapter, we'll simply want to *access* relatively small portions of a long string, in order to display them. The *slicing* technique allows you to do all of these things simply and efficiently.

We'll experiment with slicing in a series of immediate commands. Make sure you type each command into your own computer so you can actually see what slicing does.

Begin by defining the string W$, as follows:

```
LET W$ = "FUNDAMENTAL"
```

Now enter the command:

```
PRINT W$
```

to assure yourself that the word FUNDAMENTAL is actually stored in memory under the variable name W$.

The syntax for slicing uses the word TO, in parentheses, after the name of the string variable. In general, the expression:

```
S$ (M TO N)
```

identifies the portion of string S$ located in positions M to N. M and N can be literal numbers, numeric variables, or numeric expressions. We sometimes refer to the sliced portion of the string as a *substring*.

As a first try at slicing, type this command:

```
PRINT W$ (5 TO 8)
```

Before you press the ENTER key, try to predict what the result of the command will be, by counting out the positions in the word

FUNDAMENTAL. The command should display the word AMEN at the top of the screen.

You can omit the first or last position number in the TO clause. If you omit the first, the slice will begin with the first character of the string:

```
PRINT W$ (TO 3)
```

This command gives the word FUN. If you omit the last number, the slice extends to the end of the string. For example:

```
PRINT W$ (6 TO)
```

When you enter this command you should see the word MENTAL on the screen.

You can also *revise* a slice of a string. To do so, you must specify the slice that you wish to revise in a LET statement, as follows:

```
LET W$ (TO 5) = "INCRE"
```

Type this command, and then enter:

```
PRINT W$
```

to see what has happened to your string. You should see the word INCREMENTAL. The first five characters of the string have been replaced by new characters.

The decision-maker program uses the slicing technique to choose a response—from a long string of responses—in answer to a question that you will ask it. The disillusioning secret of that program is this: The responses are chosen completely at random. Before we examine the program, then, we must take a quick look at random numbers on your computer.

RANDOM NUMBERS

The function RND, located below the [T] key, returns a *random number* between 0 and 1; it returns a different number each time you use it. To see RND in action, run the following two-line program:

```
10 PRINT RND
20 GOTO 10
```

A column of random numbers will appear on the screen; the program will continue until there is no room left in the column.

Random numbers are, in principle, numbers chosen in a completely unpredictable manner. In fact, your computer uses a mathematical formula to *compute* the numbers supplied by RND. This means that the numbers are not truly random; however, the formula is designed to supply numbers that will always *seem* random. For the programs that you'll write on your computer, this *apparent* randomness will probably always be good enough.

On small personal computers such as the T/S 1000, the most common use of the random function is probably to simulate random events in games programs. For example, you can use RND to simulate a roll of dice, or a hand of randomly dealt playing cards.

To create these simulations, we need some way of converting the *range* of the random numbers generated by RND. Random numbers between 0 and 1 may not be very useful; but if we can convert them to random integers between 1 and 6, for example, then we can use the numbers in certain kinds of games.

The formula for making such a conversion is relatively simple. If you have a random number greater than 0 and less than 1, and you multiply that number by 6, you will then have a random number between 0 and 6:

```
RND * 6
```

If you truncate this number to an integer (using the INT function) you will be left with a random integer from 0 to 5:

```
INT (RND * 6)
```

Finally, adding 1 to the result, you produce a random integer from 1 to 6:

```
INT (RND * 6) + 1
```

To see this formula in action, enter and run the short program shown in Figure 5.6. This program creates a very elementary guessing game. The computer begins by displaying the following message on the screen:

```
I AM THINKING OF A NUMBER
FROM 1 TO 6. CAN YOU GUESS IT?
```

You enter a guess in the form of an integer from 1 to 6. Then the computer tells you if your guess was right or wrong. For example, if you guess 4, you'll either see a message such as:

RIGHT. THE NUMBER WAS 4.

or:

SORRY. THE NUMBER WAS 3.

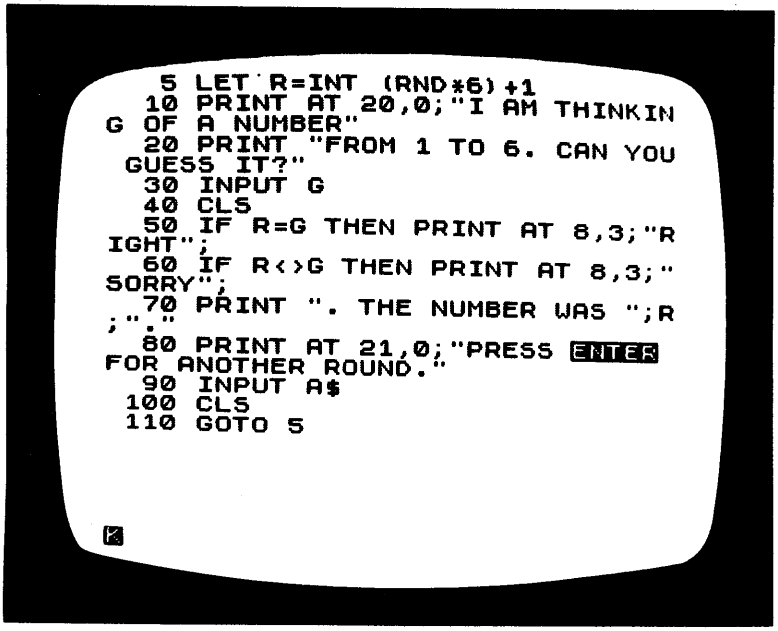
While this is a very simple game, it shows you how RND can be used to produce unpredictable results from the computer. Notice that line 5 in the program produces the random number, using the formula we developed above:

```
5 LET R = INT (RND * 6) + 1
```

You'll see a similar formula in the decision-maker program.

THE DECISION MAKER

You can use the decision-maker program as a parlor game, to amuse your friends with the computer's wonderful ability to predict



```
   5 LET R=INT (RND*6)+1
  10 PRINT AT 20,0;"I AM THINKIN
G OF A NUMBER"
  20 PRINT "FROM 1 TO 6. CAN YOU
 GUESS IT?"
  30 INPUT G
  40 CLS
  50 IF R=G THEN PRINT AT 8,3;"R
IGHT";
  60 IF R<>G THEN PRINT AT 8,3;"
SORRY";
  70 PRINT ". THE NUMBER WAS ";R
;"."
  80 PRINT AT 21,0;"PRESS ENTER
FOR ANOTHER ROUND."
  90 INPUT A$
 100 CLS
 110 GOTO 5
```

Figure 5.6: The Guessing Game Program

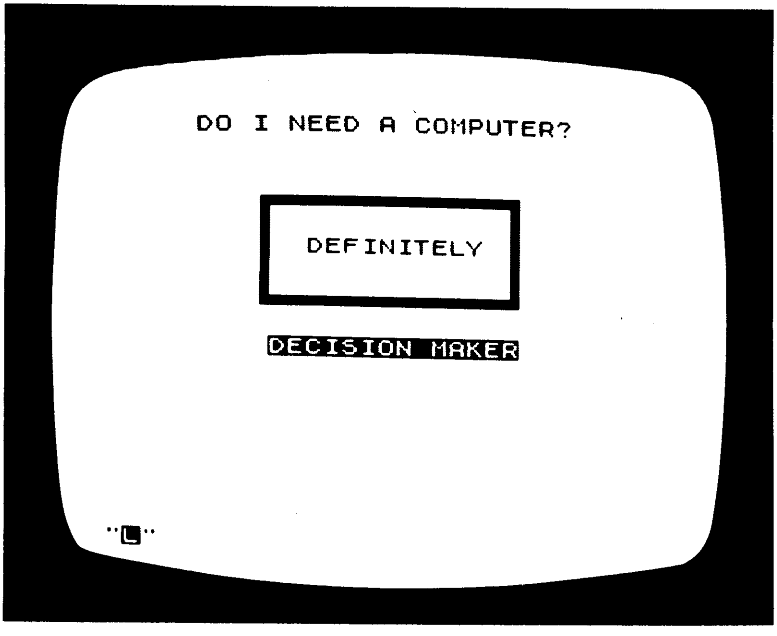
the future, solve personal dilemmas, or make fast business decisions. To operate the program, you type any yes-or-no question onto the keyboard; the computer will pause briefly to consider, and then will display its "careful" answer on the screen.

Of course, the dark secret you now share with your computer is that the answers are chosen randomly, and have no relation at all to the nature of the questions. But let's keep it a secret.

Figure 5.7 shows a sample screen from this program. After you have entered a question, the computer echoes the question at the top of the screen, and displays its answer inside the decision box. Then the computer waits for your next question; it can go on answering questions all day, if you like.

The program listing appears in Figure 5.8. We'll discuss the three important techniques illustrated in this program: the use of RND to make a random choice among several responses; the use of slicing to access the correct response; and finally, the use of the LEN function to center a string horizontally on the screen.

Begin by studying line 10 carefully. It defines the "answer



```
DO I NEED A COMPUTER?

DEFINITELY

DECISION MAKER
```

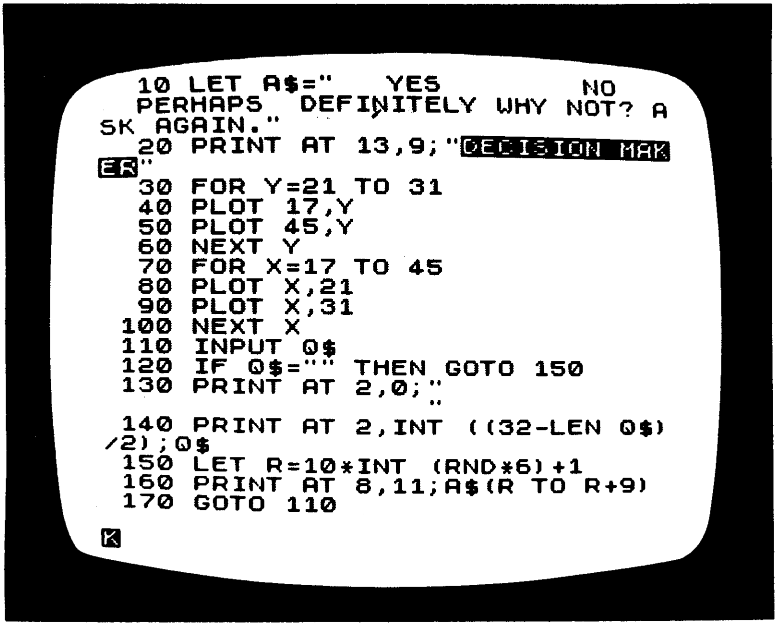
Figure 5.7: Sample Screen from the Decision Maker

string," A$. The string is exactly 60 characters long, and is made up of six 10-character sections, representing the six different answers that the computer can give to your questions:

YES
NO
PERHAPS
DEFINITELY
WHY NOT?
ASK AGAIN.

Since these answers are of varying lengths, they are surrounded by spaces in the string, which fill them out to a length of 10 characters. Thus, the first answer begins at A$(1) and goes to A$(10); the second goes from A$(11) to A$(20); the third from A$(21) to A$(30); and so on. We'll see that this regularity is the key to choosing answers from the string.

Line 20 displays the title of the program. Lines 30 to 100 draw the "decision box" on the screen, using two FOR loops and four



```
  10 LET A$="   YES        NO
  PERHAPS  DEFINITELY WHY NOT? A
SK AGAIN."
  20 PRINT AT 13,9;"DECISION MAK
ER"
  30 FOR Y=21 TO 31
  40 PLOT 17,Y
  50 PLOT 45,Y
  60 NEXT Y
  70 FOR X=17 TO 45
  80 PLOT X,21
  90 PLOT X,31
 100 NEXT X
 110 INPUT Q$
 120 IF Q$="" THEN GOTO 150
 130 PRINT AT 2,0;"
"
 140 PRINT AT 2,INT ((32-LEN Q$)
/2);Q$
 150 LET R=10*INT (RND*6)+1
 160 PRINT AT 8,11;A$(R TO R+9)
 170 GOTO 110

K
```

Figure 5.8: The Decision Maker Program

PLOT statements.

Line 110 reads your question from the keyboard and assigns it to the string variable Q$:

```
110 INPUT Q$
```

Lines 150 and 160 choose the answer. Line 150 uses RND in a formula tailored specifically for the convenience of the slicing process:

```
150 LET R = 10 * INT (RND * 6) + 1
```

We have seen that the expression INT (RND * 6) produces a random integer from 0 to 5. Multiplying this result by 10 gives us a random multiple of 10, from 0 to 50. When we add 1, we will have one of the following six numbers:

```
1   11   21   31   41   51
```

These numbers, as we have seen, represent the respective starting points of the six different answers contained in the string A$. Thus, line 150 chooses one of these starting points, at random, and assigns its value to the variable R.

Once this starting point is determined, accessing the answer is a simple slicing procedure. It is performed in line 160:

```
160 PRINT AT 8,11; A$ (R TO R + 9)
```

This line chooses the 10-character substring of A$ from position R to position R + 9; it then displays this substring on the screen, inside the decision box. So, thanks to the variable R, line 160 displays a randomly chosen answer on the screen. Line 170 then begins another round by sending control of the program back up to the INPUT statement:

```
170 GOTO 110
```

One more line in this program merits some discussion. Line 140, which displays your question at the top of the screen, contains an interesting formula for *centering* the string Q$ horizontally:

```
140 PRINT AT 2, INT ((32 - LEN Q$)/ 2); Q$
```

(If you were very observant, you might have noticed a similar formula in the bar graph program of Chapter 4.) The AT clause of this

PRINT statement puts the string on row 2; the column position is calculated in a formula based on the length of the string:

```
INT ((32 - LEN Q$)/ 2)
```

This formula assumes that the question string, Q$, is not more than 32 characters long. The LEN function supplies the number of characters in Q$. Since the screen is 32 columns wide, the expression (32 - LEN Q$)/ 2 gives the address of the column where the string must begin in order to be centered horizontally. For example, let's say you enter a question string 20 characters long; to center the string, you would want 6 spaces on both sides of the question. You can see that the formula (32 - 20)/ 2 gives you the correct starting point.

SUMMARY

Your computer codes each keyboard character as a number between 0 and 255. For most programming situations you can ignore this code; the computer uses the code quite privately, and you need not even be aware that the code exists. But for the occasions when the code can be useful to you, the functions CODE and CHR$ will help you convert between characters and code numbers.

Concatenation means combining strings to create a new string. Slicing is the process of extracting or identifying a substring. We've seen examples of both of these processes in the programs of this chapter.

At this point, you should be on good terms with your computer, its operations and its computing capabilities. You are ready to run any of the programs commercially available for your computer, or to write your own. You have a working knowledge of BASIC, a powerful and versatile programming language. In short, you are no longer merely a computer *owner,* you are a computer *user.*

Appendix A

The BASIC Vocabulary

This appendix defines the words that appear on the T/S 1000 keyboard, and, in many cases, gives examples of BASIC statements that include the words. The keyboard location of each word is specified, along with the word's category (keyword, function, or shift).

ABS (function; [G] key). Supplies the *absolute value* of a number.

AND (shift; [2] key). Creates compound true-or-false statements for IF decisions. For example:

```
IF I > 0 AND I < 100 THEN GOSUB 300
```

In this example, the GOSUB statement after THEN will only be performed if both the statements I > 0 *and* I < 100 are true.

ARCCOS (function; [S] key). Returns the arccosine of a number between −1 and 1. Result is in radians. (1 radian = 57.3 degrees.) ARCCOS is displayed as ACS on the screen.

ARCSIN (function; [A] key). Returns the arcsine of a number between −1 and 1. Result is in radians. ARCSIN is displayed as ASN on the screen.

ARCTAN (function; [D] key). Returns the arctangent, in radians. ARCTAN is displayed as ATN on the screen.

AT (function; [C] key). Used with the PRINT statement to specify a screen display position for a character, string, or digit. AT must be followed by an address in the form, *row, column,* where *row* is a number from 0 to 21, and *column* is a number from 0 to 31. For example, the statement:

```
PRINT AT 8,10; "HELLO"
```

displays the word HELLO in row 8, starting at column 10. (Address 0,0 represents the upper-left corner of the screen.)

BREAK ([SPACE] key). Interrupts a program run in progress. BREAK cannot be used while the computer is waiting for input from the keyboard.

CHR$ (function; [U] key). Supplies the character corresponding to a code number between 0 and 255. Keywords, functions, characters, and symbols are all included in the computer's character code.

CLEAR (keyword; [X] key). Clears all variables and their values from memory; does *not* clear program lines.

CLS (keyword; [V] key). The "clearscreen" command clears the screen of all information; a subsequent PRINT statement will display information starting at the upper-left corner of the screen (position 0,0).

CODE (function; [I] key). Supplies the computer's code number for a given character. Code numbers range from 0 to 255.

CONT (keyword; [C] key). Continues a program run that has been interrupted for any reason.

COPY (keyword; [Z] key). Sends the entire current contents of the screen display to the printer.

COS (function; [W] key). Supplies the cosine of a number that represents an angle. The COS function assumes that the angle is expressed in radians. (1 degree = .0175 radian.)

DELETE (shift; [0] key). Deletes the character, keyword, or function to the immediate left of the cue. DELETE can be used whenever a new line is being entered into the computer, or when a completed line is being edited.

DIM (keyword; [D] key). Defines the name, type, dimensions and length of an array. For example:

```
DIM N(10)
```

defines N, a one-dimensonal numeric array of length 10. In the case of string arrays, all the string elements must be of the same length; the length must be specified in the DIM statement. For example:

```
DIM A$(10,5)
```

defines A$, a one-dimensional string array. A$ can contain up to 10 strings, each consisting of 5 characters.

EDIT (shift; [1] key). Displays a copy of the *current line* at the bottom of the screen, ready for editing. The current line is marked with a small reverse-video " > " character in the program listing.

ENTER Enters program lines or input information into the computer's memory.

EXP (function; [X] key). Supplies the "natural exponent" of a number. (Calculates an exponent of e, where e = 2.7182818.)

FAST (shift; [F] key). Puts the computer in the fast calculation mode.

FOR (keyword; [F] key). Creates a repetition loop, causing the lines within the loop to be repeated a specified number of times. (The end of the loop is marked by a NEXT statement.) For example:

```
FOR I = 2 TO 20 STEP 2
```

The *control variable* is I in this FOR statement. During the repetition of the loop, I will be incremented in value from 2 to 20, in steps of 2.

FUNCTION (shift; [ENTER] key). Puts the keyboard into the function mode; the next keystroke will register as a function key.

GOSUB (keyword; [H] key). Sends control of the program to a subroutine. The first line of the subroutine can be expressed as a literal number:

```
GOSUB 300
```

or as a numeric variable:

```
GOSUB T
```

or even as a numeric expression, for a "calculated" GOSUB:

```
GOSUB 100 * (VAL I$ + 4)
```

GOTO (keyword; [G] key). Sends control of the program to a specified line. The line number can be expressed as a literal numeric value, a numeric variable, or a numeric expression. (See GOSUB for examples.)

GRAPHICS (shift; [9] key). Puts the keyboard into the graphics mode; subsequent keystrokes will register as graphics characters or reverse-video characters. The GRAPHICS key must also be used to shift the keyboard out of the graphics mode back into the letter mode.

IF (keyword; [U] key). Introduces a decision statement. An IF statement begins with a true-or-false statement, followed by a THEN clause; for example:

```
IF D <> 0 THEN LET Q = 1 / D
```

With this statement, the computer first evaluates D <> 0 to true or false. If the statement is true, the LET statement (after THEN) is performed; if it is false, control of the program proceeds to the following line.

**INKEY$** (function; [B] key). For each INKEY$ performance, the computer scans the entire keyboard once to see if a key is being pressed. If so, INKEY$ supplies the letter-mode value of the key; if not, INKEY$ returns a null string ("").

INPUT (keyword; [I] key). Instructs the computer to read data from the keyboard and to store the data in a specified variable. For example:

```
INPUT S$
```

reads a string input from the keyboard, and stores the value in the variable S$. INPUT causes the input data to be echoed on the screen as it is entered.

INT (function; [R] key). Supplies the integral value of a number.

LEN (function; [K] key). Supplies the length, in characters, of a string.

LET (keyword; [L] key). Assigns a value to a variable. The variable can be either a new one, receiving its first value, or an old one, receiving a new value. At the left side of the equal sign, the LET statement must contain a single variable name; the right side may contain literals, variables, or expressions. For example:

```
LET V = 5
LET A = B
LET X = (5 * Y) / 2
```

Any variables named on the right side of the equal sign must be defined before the execution of the LET statement. The LET statement does not change the values of variables on the right side of the equal sign.

LIST (keyword; [K] key). Displays the current program on the screen. If LIST is followed by a line number, as in:

```
LIST 100
```

the program listing begins at that line.

LLIST (shift; [G] key). Sends the program listing to the printer.

LN (function; [Z] key). Supplies the natural logarithm (base e) of a positive number.

LOAD (keyword; [J] key). Retrieves a program stored on cassette tape. The LOAD command allows two forms: with the name of the program:

```
LOAD "NAME"
```

or without the name:

```
LOAD ""
```

The quotation marks are required in both forms. In the latter case, the computer loads the first program it encounters on the cassette tape.

LPRINT (shift; [S] key). Sends a line of information to the printer.

NEW (keyword; [A] key). Clears the current program out of the computer's memory.

NEXT (keyword; [N] key). Marks the end of a FOR loop. The control variable, defined in the FOR statement, must follow the word NEXT; for example:

```
NEXT I
```

NOT (function; [N] key). Modifies a true-or-false statement in an IF decision; for example:

```
IF NOT D > 0 THEN GOTO 10
```

This statement is equivalent to:

```
IF D< =0 THEN GOTO 10
```

OR (shift; [W] key). Creates compound true-or-false statements for IF decisions. In the following example, OR connects two equalities:

```
IF X=5 OR Y=3 THEN GOSUB 300
```

The GOSUB statement expressed after the word THEN will be performed if either or both of the equalities are true.

PAUSE (keyword; [M] key). Instructs the computer to pause in the execution of a program. The length of the pause is determined by an integer following the keyword PAUSE. For example:

```
PAUSE 50
```

results in a pause of about one second; increasing the number by multiples of 50 will increase the pause by additional seconds. If the number following PAUSE is greater than 32766, the command means "pause forever." Any pause can be interrupted, and the program resumed, by pressing any key on the keyboard.

PEEK (function; [O] key). Returns the contents of a memory location from 0 to 65535.

PLOT (keyword; [Q] key). Displays a "pixel" on the screen at a specified address. The elements of the PLOT address are in the reverse order of the elements of the PRINT AT address. The general form of the PLOT command is:

PLOT *h,v*

where *h* is the horizontal address (from 0 to 63) and *v* is the vertical address (from 0 to 43). The PLOT address 0,0 is located at the lower-left corner of the screen.

POKE (keyword; [O] key). Writes a value to a specified memory location from 0 to 65535. Takes the form:

```
POKE a,v
```

where *a* is the address of the memory location and *v* is the value. The absolute value of *v* cannot exceed 255.

PRINT (keyword; [P] key). Sends a line of information to the screen. A semicolon (;) separating PRINT elements displays the elements side-by-side, with no space between them. A comma (,) between elements results in a tab forward to the center of the screen or to the beginning of the next screen line, whichever is closer.

RAND (keyword; [T] key). Controls the starting point of a sequence of random numbers produced by the RND function. The statements:

```
RAND
```

and:

```
RAND 0
```

result in an unpredictable starting point. The statement:

```
RAND n
```

where *n* is greater than 0 and less than 65536, results in a specific and predictable starting point.

REM (keyword; [E] key). Creates a comment (or "remark") line in the program listing. The computer ignores REM lines during the program run. Any kind of comment can be placed after the word REM.

RETURN (keyword; [Y] key). Indicates the end of a subroutine. Sends control of the program back to the line *following* the GOSUB statement that originally called the subroutine.

RND (function; [T] key). Produces a "random" number greater than or equal to zero, and less than 1.

RUN (keyword; [R] key). Instructs the computer to begin executing the lines of the program currently held in memory. (Begins by performing a CLEAR of any variables from previous program runs.) If RUN is followed by a program line, the computer begins the program run at the indicated line.

SAVE (keyword; [S] key). Stores a program onto a cassette tape. The SAVE command requires a program name, within quotation marks:

```
SAVE "NAME"
```

SCROLL (keyword; [B] key). Moves all the information displayed on the screen up by one line. The top line is lost, and an empty line appears at the bottom of the screen.

SGN (function; [F] key). Supplies the *sign* of any number; results in a value of −1, 0, or 1.

SLOW (shift; [D] key). Returns the computer to the slow calculation mode.

SPACE Places a space character in a program line or an input string.

SQR (function; [H] key). Supplies the square root of a nonnegative number.

STEP (shift; [E] key). In a FOR statement, indicates the incrementation (or decrementation) amount for the control variable. For example:

```
FOR I = 5 TO 25 STEP 5
```

In this statement the control variable I will be increased by 5 for each repetition of the loop. If the STEP clause is missing from a FOR statement, the default incrementation value is 1.

STOP (shift; [A] key). Stops execution of a program. The word STOP may be part of a program line, or it can be input from the keyboard to end a program run.

STR$ (function; [Y] key). Supplies the string version of a number. For example:

```
LET S$ = STR$ 1234
```

This statement will result in the string value "1234" being stored in the variable S$.

TAB (function; [P] key). Used with the PRINT statement to display a message beginning at a specified column of the current line. For example:

```
PRINT TAB 15; "HELLO"
```

will print HELLO beginning at column 15.

TAN (function; [E] key). Supplies the tangent of a number that represents an angle. TAN assumes that the angle is in radians (1 degree = .01745 radians).

THEN (shift; [3] key). Introduces the result of an IF statement. THEN is always followed by a keyword. The statement after THEN is performed only if the statement after IF is evaluated to true.

TO (shift; [4] key). Introduces the maximum value of the control variable in a FOR loop. For example:

```
FOR I = 1 TO 10
```

TO is also used for "slicing" strings. The following LET statement assigns values to the 2nd through 4th positions in the string S$:

```
LET S$(2 TO 4) = "ABC"
```

UNPLOT (keyword; [W] key). Erases a pixel from the screen at a specified address. The general form of the UNPLOT command is:

UNPLOT *h,v*

where *h* is the horizontal address (from 0 to 63) and *v* is the vertical address (from 0 to 43). The UNPLOT address 0,0 is located at the lower-left corner of the screen.

USR (function; [L] key). Calls a machine-language routine located at a specified memory address; the general form of USR is:

USR *a*

where *a* is the memory address of the beginning of the routine.

VAL (function; [J] key). Supplies the numeric value of a string. The characters of the string must consist of digits, functions, calculations, or variables that form a valid numeric expression.

T/S 1000 Error Codes

At the end of every program run, a message appears in the lower-left corner of the screen in the form:

```
m/n
```

where *n* is the line number where program execution stopped, and *m* is one of the following error codes:

0	No error; program completed.
1	Discrepancy between control variables named in FOR and NEXT statements.
2	Use of an undefined variable name.
3	Array index is outside of the range specified in DIM statement.
4	Computer is out of memory.
5	Out of room on TV screen.
6	A calculation has resulted in a number too large for the computer to handle.
7	RETURN statement not preceded by GOSUB.
8	INPUT used as an immediate command.
9	Program ended on a STOP statement.
A	Illegal use of a function (for example: SQR − 1 or LN 0).
B	Illegal address, code number, or other integer in statements such as PRINT AT, PLOT, CHR$, etc.
C	The string expression after VAL cannot be evaluated to a number.
D	Program interrupted from keyboard by BREAK or STOP keys.
E	Not used.
F	SAVE command without program name.

Loading [illegible]

Save "program name"

push cassette [illegible] [illegible]

Red plugs in MIC - Vol. turned down slightly from mid

Load "program name"

push cassette - enter

Black plug in ear - Vol turned up slightly from mid

Index

YOUR TIMEX SINCLAIR 1000™ AND ZX81™

Your Timex Sinclair 1000™ and ZX81™

takes you from the very beginning and explains in simple, everyday language how to use your Timex Sinclair 1000 or ZX81 to its fullest capabilities.

You will quickly learn how to . . .

- connect your TV and cassette recorder to the computer and make them work together
- use the keyboard to give commands to your computer
- write your own programs for graphics, calculations, games, and more
- use the easy and versatile BASIC programming language
- use the ready-to-run programs included in this book to:
  —turn your computer into a super calculator
  —make bar graphs to help calculate home finances
  —draw pictures on your TV screen

Your Timex Sinclair 1000 and ZX81 will help you get the most out of your new computer.

ABOUT THE AUTHOR:

Douglas Hergert is on the Sybex editorial staff, and is the author of *BASIC for Business* and *Mastering VisiCalc.* Drawing on his broad computer expertise, he has also contributed to several other Sybex titles as editor and translator.

ISBN 0-89588-099-7